Formation R

Benoît Lepage

2025-08-29

Contents

Chapter 1

Bienvenue sur cette formation au logiciel R

R est un logiciel accessible gratuitement permettant de réaliser des analyses statistiques dans un environnement windows, macOS ou Linux.

1.1 Pourquoi choisir R ?

Le logiciel est gratuit, très complet, avec une communauté d'utilisateurs très active dans le monde entier. Il est fréquent que les nouvelles méthodes d'analyses statistiques développées dans les équipes académiques soient d'abord mises à disposition sur R.

Le logiciel R repose sur l'utilisation de **scripts** dans lesquels nous allons **programmer** les analyses statistiques. Cette écriture sous forme de programmation peut paraître austère à première vue, mais est indispensable pour permettre la **reproductibilité** et la **transparence** des analyses. La même démarche de programmation est utilisée dans tous les logiciels statistiques professionnels (Stata, SAS, Python, Matlab, etc).

Pour utiliser R, les premières choses à faire sont de :

- télécharger le logiciel R
- et télécharger un Environnement de Développement Intégré (IDE) comme RStudio.

1.2 Téléchargez le logiciel R

Vous pouvez télécharger la dernière version stable du logiciel R sur le site du R project.

Cliquez sur "download R", choisissez un site mirroir (par exemple un des sites en France).

Puis téléchargez la version de R en fonction de votre système d'exploitation (Windows, macOS ou Linux).

Enfin, installez R à partir du fichier d'installation que vous venez de télécharger.

1.2.1 Ouvrez le logiciel R

Si vous ouvrez le logiciel R, vous aller trouver l'interface graphique de R (*RGui* pour *R Graphical user interface*). Il est possible de faire vos analyses statistiques à partir de cette interface graphique, mais elle est très très austère.

Plutôt que d'utiliser cette interface RGui, nous vous recommandons fortement d'utiliser un Environnement de Développement Intégré (IDE), comme RStudio, qui vous facilitera grandement la vie pour utiliser un logiciel statistique qui repose sur de la programmation.

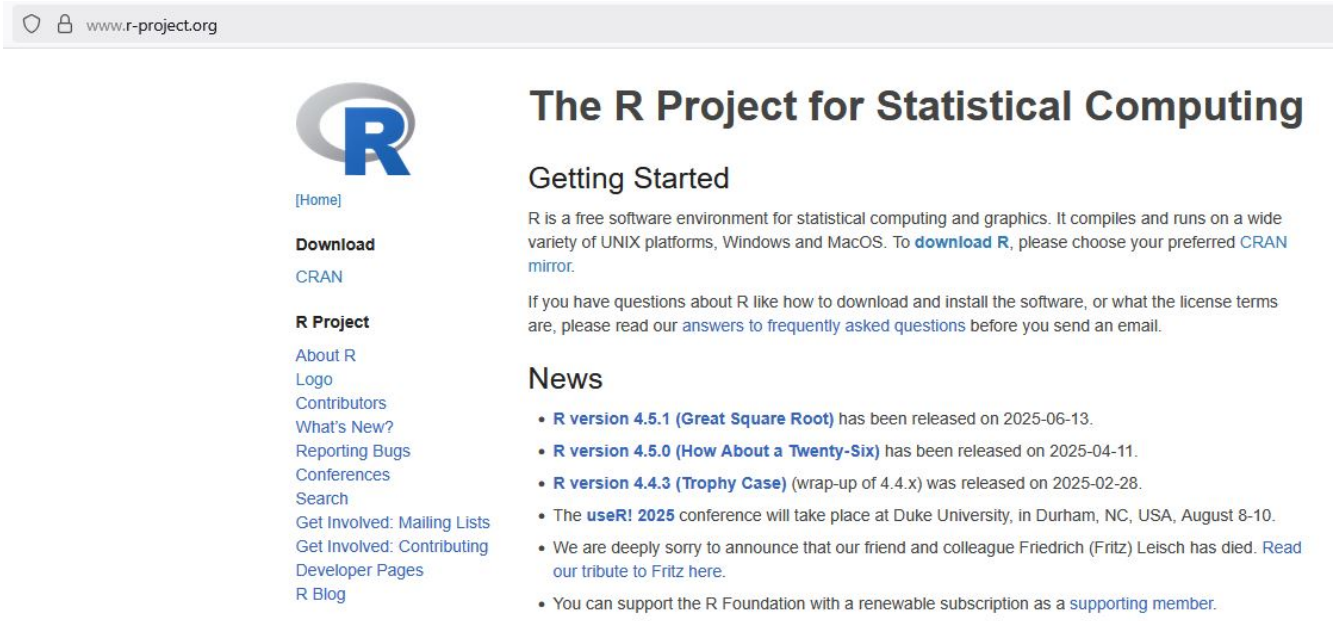


Figure 1.1: Site du R project, en juillet 2025

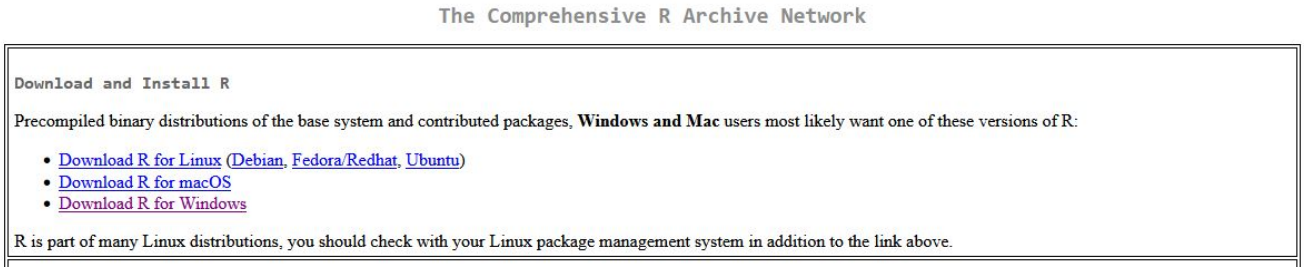


Figure 1.2: Choisissez la version adaptée à votre système d'exploitation

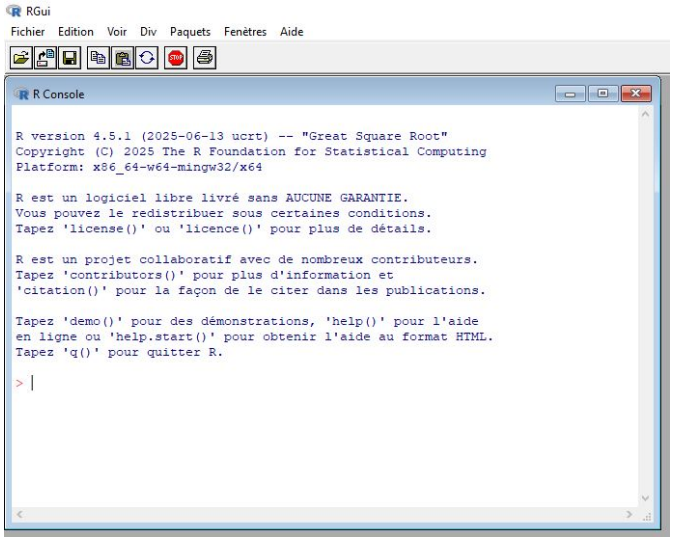


Figure 1.3: L'interface graphique de R (RGui)

1.3 Téléchargez un IDE (RStudio recommandé)

RStudio est un environnement qui permet d'utiliser R, mais également d'autres logiciels de programmation comme Python, SQL, Stan, C++, etc. Cet environnement vous facilitera le travail pour :

- éditer vos scripts de programmation,
- accéder à la console,
- visualiser vos environnements de travail avec les fichiers et les objets qu'il contient,
- visualiser vos sorties graphiques et certaines tables d'analyses,
- visualiser vos données,
- visualiser les fichiers d'aide,
- gérer les *packages* permettant de faire des analyses spécifiques,
- et bien d'autres choses encore.

Par exemple, le tutoriel que vous êtes en train de lire a été créé à partir du package `bookdown` avec le logiciels R, au sein de l'IDE RStudio,

Vous pouvez télécharger la dernière version de RStudio sur le site de la compagnie Posit. Choisissez la version qui est adaptée à votre système d'exploitation (Windows, macOS ou Linux).

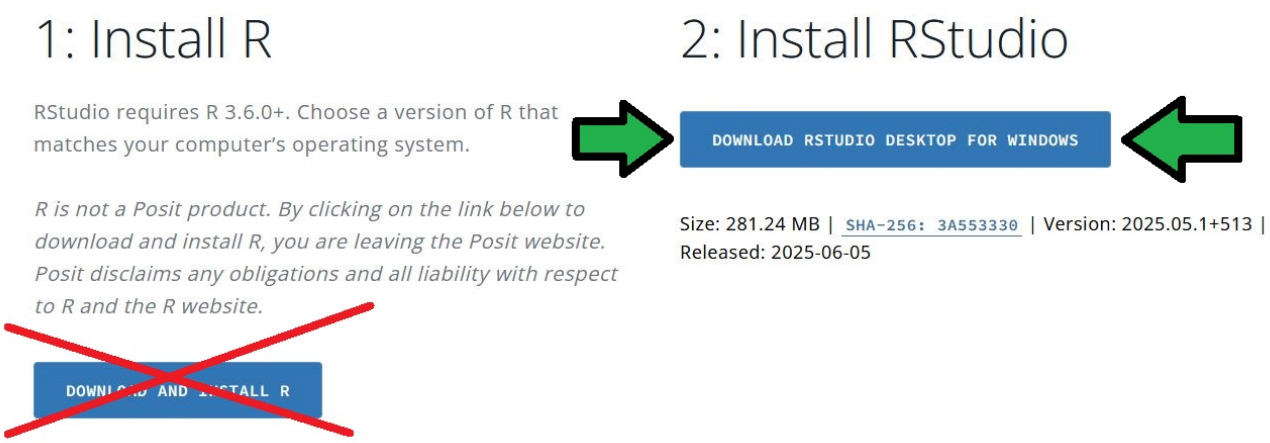


Figure 1.4: téléchargez RStudio

Puis, installez RStudio à partir du fichier d'installation que vous venez de télécharger.

1.3.1 Ouvrez l'IDE RStudio

Ouvrez RStudio, puis commencez par ouvrir un **script**

- à partir du menu File > New File > R script
- ou bien en utilisant le raccourci Ctrl+Maj+N sur windows
- ou bien en cliquant sur le petit fichier blanc avec un + vert en haut à gauche, puis choisir "R script"

L'interface de RStudio contient un menu, 4 quadrants et des sous-menus et boutons dans chaque cadrant.

Les menus qui vous seront le plus utiles sont :

- Dans le menu principal,

OS	Download	Size	SHA-256
Windows 10/11	RSTUDIO-2025.05.1-513.EXE	281.24 MB	3A553330
macOS 13+	RSTUDIO-2025.05.1-513.DMG	607.30 MB	76E1538B
Ubuntu 22/Debian 12	RSTUDIO-2025.05.1-513-AMD64.DEB	209.78 MB	89A68B37
Ubuntu 24	RSTUDIO-2025.05.1-513-AMD64.DEB	209.78 MB	89A68B37
Fedora 41	RSTUDIO-2025.05.1-513-X86_64.RPM	224.98 MB	6A97DF24

Figure 1.5: téléchargez RStudio

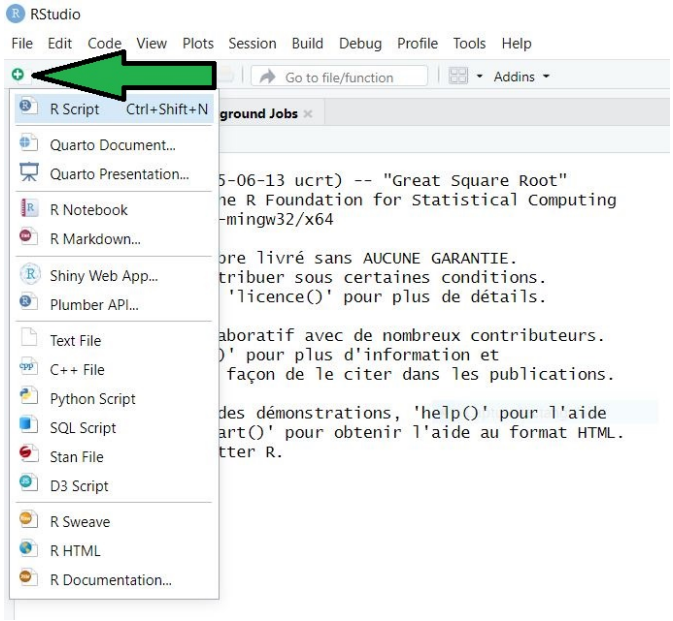


Figure 1.6: Ouvrir un nouveau script

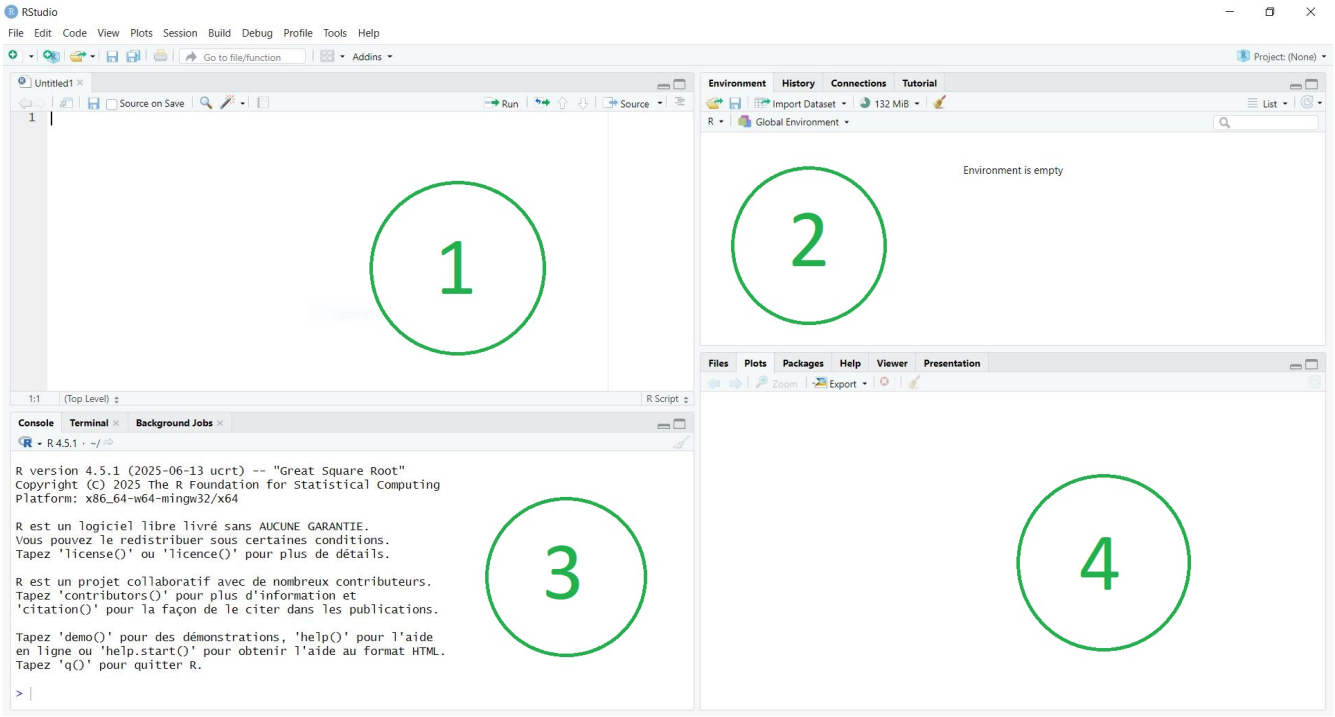


Figure 1.7: Les 4 cadrants de RStudio

  - le menu *File* vous permettra de créer de nouveaux fichiers, d'ouvrir des fichiers déjà existants, de sauver vos fichiers, d'importer des bases de données, etc.
  - le menu *Tools* > *Install packages…* pour installer de nouveaux packages
  - le menu *Tools* > *Global Options…* vous permet de choisir la version du logiciel R à utiliser (onglet "R General") ou bien de changer l'aspect graphique de l'environnement RStudio (onglet "Appearance", puis choisissez un "Editor theme", avec différentes interfaces claires ou sombres)

- Au sein du **script** (cadrant 1)
  - le bouton "disquette" permet de sauvegarder votre script
  - le bouton "run" permet de faire tourner votre programme d'analyse (les lignes que vous avez sélectionnées). Par exemple, tapez la commande suivante dans le script, sélectionnez la ligne et cliquez sur le bouton "run" (ou avec un raccourci clavier `ctrl`+`entrée` sur windows, ou encore `command`+`entrée` sur macOS).

```
print("Hello Toulouse")
```

et vous devriez voir la commande `> print("Hello Toulouse")` puis son résultat `"Hello Toulouse"` dans l'onglet **console** du cadrant 3.

- Au sein du cadrant 3, l'onglet le plus utile pour pour les débutants est l'onglet **console**
  - la console est la même que la console affichée dans l'interface RGui du logiciel R que l'on a vu au paragraphe 1.2.1.
  - la console commence par afficher la version de R en cours d'utilisation
  - vous pouvez y saisir des commandes et obtenir directement leurs résultats, par exemple si vous tapez dans la console `4+9`, vous obtiendrez directement le résultat `13`. **Attention, les commandes que vous saisissez directement dans la console ne seront pas sauvegardées. Si vous voulez sauvegarder des commandes, il faut utiliser le *script* (cadrant 1)**

```
4+9
```

```
## [1] 13
```

- Au sein du cadrant 2, l'onglet le plus utile pour les débutants est l'onglet **Environment**
  - cet onglet vous permettra de visualiser les "objets R" créés pendant vos analyses.
  - Par exemple si vous saisissez `v <- 1:10` dans la console, vous allez voir apparaître l'objet `v` dans l'environnement de travail (il s'agit d'un vecteur de 1 à 10, nommé "v").
- Au sein du cadrant 4, les onglets les plus utiles pour les débutants sont :
  - l'onglet "File" qui contient les dossiers et fichiers au sein d'un dossier de travail (voir le chapitre 3 pour créer et organiser un dossier de travail associé à un "projet R")
  - l'onglet "Plots" où vous retrouverez vos sorties graphiques. Au sein de cet onglet, vous trouverez un menu pour exporter vos graphiques selon différents formats. Des boutons permettent également de zoomer et d'effacer les graphiques. Par exemple, si vous saisissez `hist(rnorm(10000))` dans la console, un histogramme d'une distribution normale centrée réduite va apparaître. Vous pouvez effacer la figure en cliquant sur le bouton avec la croix rouge (efface la figure actuelle) ou le balet (efface l'ensemble des figures).
  - l'onglet "Packages" où vous pourrez activer, désactiver ou mettre à jour les packages qui ont été téléchargés.
  - l'onglet "Help" où vous trouverez de l'aide. Par exemple si vous saisissez `help(mean)` dans la console, l'aide de la commande `mean` va s'afficher. Vous pouvez également utiliser le champ de recherche de fonctions dans le menu "Help".

1.4 Trouver de l'aide sur R

De nombreuses ressources sont disponibles pour vous aider à utiliser R :

- Les pages d'aide en ligne de R, qui apparaîssent directement dans RStudio. Vous pouvez obtenir de l'aide sur des fonctions et des packages :
  - en appliquant une recherche par mot clé dans le champ de recherche de l'onglet "help"
  - en utilisant directement dans la console la fonction `help.search()` ou `??` associée à un mot clé (par exemple `help.search(student)` ou `??student`), ou la fonction `help()` associée à une fonction (par exemple `help(t.test)` ou `?t.test`)
  - quand vous rencontrerez une nouvelle fonction (dans ce cours ou bien dans votre pratique), nous vous conseillons d'aller systématiquement regarder l'aide de cette fonction pour bien comprendre comment l'utiliser.
  - Ces pages d'aide suivent la structure suivante :
    - une partie "Description" qui décrit en quelques phrases ce que fait la fonction
    - une partie "Usage" qui décrit la syntaxe de la fonction avec ses arguments
    - une partie "Argument" qui précise comment renseigner les arguments de la fonction
    - une partie "Detail" qui décrit en détail comment utiliser la fonction et ses arguments
    - une partie "Value" qui décrit les sorties (les résultats) de la fonction, avec les éventuels sous-objets de la sortie
    - une partie "Exemples" qui indique quelques exemple que vous pouvez directement lancer en cliquant sur "Run examples"
- Des fiches "mémoires" *cheat sheets* qui résument les principales fonctions :
  - pour les commandes R bases
  - des bonnes pratiques sur R
  - méthodes de visualisation avec le package ggplot2
  - méthodes de manipulation de données avec le package tidyr

  - méthodes de transformation de données avec le package dplyr
  - l'utilisation du package data.table
  - l'utilisation du package stringr pour manipuler les chaînes de caractères, avec la fiche stringr
  - la manipulation de dates Dates and times with lubridate et la fiche lubridate

- De nombreux livres et tutoriels disponibles gratuitement en ligne :

  - Le guide R de Joseph Larmarange, est un guide très complet et didactique, en français
  - L'Epidemiologist R Handbook est un tutoriel en anglais pour l'utilisation de R par des épidémiologistes
  - Software carpentry met à disposition des guides introductifs bien réalisés en anglais, par exemple R for Reproducible Scientific Analysis ou encore Programming with R
  - le livre R for Data Science propose une introduction très complète pour l'analyse de données descriptive principalement basée sur la suite de packages du Tidyverse
  - le package R swirl propose une formation interactive directement dans la console de RStudio, téléchargez le package (`install.packages("swirl")`), chargez le package (`library("swirl")`) et laissez vous guider après avoir saisi `swirl()` dans la console.
  - RStudio Education liste plusieurs ressources intéressantes pour les débutants
  - Le site CRAN a des manuels assez complets, par exemple la page R Language Definition est une introduction assez complète au langage de programmation R. La page CRAN Task Views liste les packages qui sont disponibles par thématique ou type d'anlayse.

- Rechercher à l'aide d'un moteur de recherche (google, DuckDuckGo, Bing, etc). Ces recherches vous amèneront régulièrement vers des forums de discussion comme stackoverflow ou stackExchange. Si vous rencontrez une erreur ou une difficulté, il y a toutes les chances que d'autres personnes aient déjà rencontré ces erreurs et difficultés avant vous, et que des solutions détaillées soient proposées dans ces forums.

- Les chatbots de type GPT, Copilot ou Gemini : dans le cadre de votre formation au logiciel R, nous vous déconseillons l'utilisation de ces outils basés sur des LLM. Ces outils posent de nombreux problèmes en termes de transparence, de respect de droit d'auteur, d'impact environnemental, de déqualification (délegation de compétences pour rechercher de l'information, perte d'esprit critique), de dépendance aux Gafams, de dégradation des systèmes d'information, etc. Par ailleurs, bien qu'ils peuvent apporter des solutions fonctionnelles, il est bien plus utile d'avoir une bonne compréhension des bases de programmations sur R avant d'utiliser de tels outils : sans une bonne compréhension de la logique de programmation et des principales fonctions de R, vous aurez des difficultés à évaluer la fiabilité des solutions proposées, à vous débloquez en cas de problèmes, ou encore à adapter vos prompts pour obtenir de meilleures réponses. Les ressources décrites précédemment devraient vous permettre d'apporter efficacement des réponses à vos questions.

1.5 Conventions d'écriture

Certaines conventions ont été proposées pour faciliter l'écriture et la lecture du code de programmation dans R. Elles ne sont pas obligatoires, mais nous vous encourrageons à les suivre.

Par exemple, le *Tidyverse style guide* :

- nommer les variables et les fonctions en lettres minuscules, à l'aide de mots et de chiffres séparés par `_` (*underscore*). Par exemple `csp_1`. Cette convention fait référence au style "snake case"
- ajouter un espace après une virgule, par exemple `x[2, 5]`
- ajouter des espaces avant et après les opérateurs arithmétiques, par exemple `x <- (1 + 2) / 5`, à quelques exceptions près (pas d'espace avant ou après le signe "puissance" `^`) `y <- x^2 + 3`

Chapter 2

Les objets dans R

La programmation R repose sur des *objets*, qui apparaîtront dans la fenêtre *Environment* de RStudio.

Les objets les plus élémentaires dans R sont des **vecteurs**. Un vecteur contient une série de valeurs (des nombres, des chaînes de caractères, ou des données plus complexes). Il y a deux types de vecteurs :

- les **vecteurs atomiques** (les valeurs d'un vecteur atomique doivent toujours être du même type),
- les **listes** (les valeurs d'une liste peuvent être de différents types).

2.1 Manipuler les objets dans l'environnement

Voici quelques commandes de gestion des objets dans votre environnement :

- dans la console, commencez par créer les objets suivants. Pour **assigner une ou plusieurs valeurs à un objet**, on utilise une flèche dirigée vers la gauche `<-`. Vous verrez apparaître ces objets dans la fenêtre *Environment*.

```
## note : le signe dièze (#) permet d'ajouter des commentaires dans le code
## - le 1er objet est un vecteur de 10 nombres entiers de 1 à 10
## - le 2ème objet est un vecteur de 3 lettres A, B et C
## - le 3ème objet est un vecteur de 2 réels, calculés par 2 opérations
## - le 4ème objet est une fonction qui ajoute 2 aux éléments du vecteur x
## - le 5ème objet est un scalaire égal à 42
objet_1 <- c(1:10)
objet_2 <- c("A", "B", "C")
objet_3 <- c(10 / 3, 4 * 5)
objet_4 <- function(x) {x + 2}
objet_5 <- 42
```

- la commande `ls()` permet de **lister** les objets dans l'environnement.
- la commande `rm()` permet de **supprimer** (*remove*) un ou plusieurs objets de l'environnement.

```
ls()
```

```
## [1] "objet_1" "objet_2" "objet_3" "objet_4" "objet_5"
```

```
rm(objet_2, objet_5)
rm(list = ls()) # pour supprimer tous les objets présents dans l'environnement
```

Conseils pour nommer un objet dans R :

- Utiliser des noms courts, mais lisibles, faciles à comprendre
- Les noms doivent commencer par une lettre (pas par un nombre) et ne peuvent pas contenir d'espace. A noter qu'ils peuvent contenir des points, comme par exemple `nom.variable`.
- Evitez d'utiliser des noms de variables et de fonctions déjà existants dans R (par exemple `mean` qui risque de porter à confusion avec la fonction `mean()` : il vaut mieux utiliser `mean_variable`)
- Ecrire en minuscule avec underscore pour séparer les mots (par exemple `date_naissance`)

2.2 Principaux types de données

Les données peuvent être de différents types. Les 4 principaux types sont :

- les **nombres réels** (`?double`), par exemple `12.43`.
- les **nombres entiers** (`?integer`). Les nombres entiers sont saisis en ajoutant `L` à droite du nombre, par exemple `5L`.
- les **chaînes de caractères textuels** (`?character`), définis avec des guillemets simples ou doubles , par exemple `'bonjour'` ou `"au revoir"`
- les **valeurs logiques** (`?logical`), avec deux valeurs possibles :
  - valeur booléenne *vraie*, notée `TRUE` ou bien `T`
  - valeur booléenne *fausse*, notée `FALSE` ou bien `F`

On peut également trouver des types de données un peu plus sophistiquées, construites à partir des principaux types :

- des **variables qualitatives** qui peuvent être **nominales** (`?factor`) ou **ordinales** (`?ordered`). Ces variables sont construites sur des nombres entiers.
- des **dates** (`?Date`), qui sont construites sur des nombres réels.
- des **dates-heure** (`?POSIXct`), qui sont construites sur des nombres réels.
- des **durées** (`?difftime`), construites sur des nombres réels.

On a également un vecteur particulier qui est le **vecteur nul** et se note `NULL`. Le vecteur nul a une longueur de 0 et ne peut avoir aucun attribut (les notions de longueur et d'attribut d'un vecteur seront vues plus bas).

2.2.1 Données manquantes

Quel que soit le type de données, les données manquantes se notent `NA` (not applicable).

Attention à ne pas confondre le vecteur nul `NULL` et les données manquantes `NA`.

2.2.2 Décrire le type de l'objet ♠

Les paragraphes avec un ♠ présentent des notions plus avancées, si vous êtes en phase d'apprentissage, vous pouvez aller directement au paragraphe suivant.

On peut décrire quel est le type de l'objet avec les fonctions `typeof` (le type le plus élémentaire), `mode` et `storage.mode` (mode de l'objet et mode de stockage de l'objet selon un regroupement un peu plus large).

Par exemple, les nombres réels (`double`) et les nombres entiers (`integer`) sont du mode `numeric`.

```r
## les valeurs réelles ('double') et les entiers ('integer') sont de mode 'numeric'
typeof(2.53) # un réel
typeof(5L) # et un entier
mode(2.53) # sont de mode 'numeric'
mode(5L)
storage.mode(2.53)
storage.mode(5L)

## les chaînes de caractères sont de type et de mode 'character'
typeof(c("hello", "Toulouse"))
mode(c("hello", "Toulouse"))

## les valeurs logiques sont de type 'logical'
typeof(c(TRUE, FALSE, FALSE))
mode(c(TRUE, FALSE, FALSE))
```

x	typeof(x)	mode(x)	storage.mode(x)
2.53	"double"	"numeric"	"double"
5L	"integer"	"numeric"	"integer"
"bonjour"	"character"	"character"	"character"
TRUE	"logical"	"logical"	"logical"
as.Date("2025-07-01")	"double"	"numeric"	"double"

Les fonctions `as.double`, `as.integer`, `as.character`, `as.logical` permettent de forcer par coercition le type d'un objet *en tant que* réel, entier, chaîne de caractères, logique.

```r
as.double(5L) # définit un nombre entier en tant que nombre réel
as.integer(4.95) # définit un réel en tant qu'entier, seul l'entier est conservé
as.character(4.95) # définit un nombre en tant que chaîne de caractères

## définir une valeur logique TRUE et FALSE en tant que valeur numérique
## ou en tant qu'entier donne les valeurs 1 et 0, respectivement
as.numeric(TRUE)
as.numeric(FALSE)

## définir le nombre 0 en tant que valeur logique donne la valeur FALSE
as.logical(0)

## définir tout nombre différent de 0 en tant que valeur logique
## donne la valeur TRUE
as.logical(-14)
as.logical(1)
as.logical(4.95)
```

Les fonctions `is.double`, `is.integer`, `is.character`, `is.logical` permettent d'évaluer si un objet est de type réel, entier, textuel, logique.

```r
is.double(5L) # FALSE, un nombre de type "entier" n'est pas de type "réel"
              # attention, en math, les nombres entiers font partie des réels !
is.integer(4.95) # FALSE, un nombre de type "réel" n'est pas de type "entier"
is.numeric("bonjour") # FALSE "bonjour" est une chaîne de caractères
is.character("bonjour") # TRUE, "bonjour" est bien une chaîne de caractères
is.character(4.95) # FALSE, 4.95 est un objet numérique
```

```r
is.logical(1) # FALSE, 1 est un objet numérique
is.logical(as.logical(1)) # TRUE, as.logical(1) = TRUE, qui est un objet logique
is.logical(TRUE) # TRUE est bien un objet logique
```

2.3 Principales structures de données

Les principales structures de données que nous allons détailler dans la suite de ce chapitre sont :

- les **vecteurs atomiques** (`?c()`), qui doivent toujours comporter des valeurs du même type. Les vecteurs qui ne comportent qu'une seule valeur sont appelés des "scalaires" ;
- les **listes** (`?list`), qui sont également des vecteurs, mais peuvent comporter des valeurs de types différents ;
- les **matrices** (`?matrix`, `?array`), qui sont des vecteurs atomiques réarrangés sous forme de tables à 2 dimensions ou plus ;
- les **bases de données** (`?data.frames`). Les bases de données sont des listes de vecteurs atomiques de même longueur. Il existe d'autres formats de base de données qui seront présentés plus tard (avec les packages `tidyverse` et `data.table`).

2.4 Objet à une seule valeur (scalaire ou texte)

2.4.1 Scalaires

Assignez les valeurs 4 et 5 à deux objets

```r
x_1 <- 4
x_2 <- 5
```

2.4.2 Opérations mathématiques sur les scalaires

2.4.2.1 Calculatrice

On peut utiliser les opérations classiques, comme sur une calculatrice :

- `+` pour **additionner**
- `-` pour **soustraire**
- `*` pour **multiplier**
- `/` pour **diviser**
- `^` pour mettre à la **puissance**
- `e` pour la **notation scientifique**

```r
x_1 + x_2 # 4 + 5 = 9
10 - x_1 # 10 - 4 = 6
x_1 * x_2 #  4 * 5 = 20
20 / x_2 # 20 / 5 = 4
x_1^2 #  4^2 = 16
10^-1 # 1/10 = 0.1
25^(0.5) # racine carrée de 25 (puissance 1/2)

## notation scientifique pour les grands et petits nombres
1/1000000 # 1 pour 1 million = 1e-6
1/1e6
1e6 * 1000 # 1 million * 1000 = 1 milliard
```

2.4.2.2 Fonctions mathématiques

Plusieurs fonctions mathématiques de bases sont implémentées nativement dans R :

- `log(x)` ou `log(x, base = exp(1))` pour le **logarithme** népérien,
- `log10(x)` pour le logarithme base 10, `log2(x)` pour le logarithme base 2,
- `log(x, base = b)` pour le logarithme base `b`,
- `exp(x)` pour l'**exponentielle** de `x`
- `sqrt(x)` pour la **racine carrée** de `x`
- `abs(x)` pour la **valeur absolue** de `x`
- les **fonctions trigonométriques** sont implémentées, avec `cos(x)`, `sin(x)`, `tan(x)` (cf. `?Trig`)
- la **constante** π est implémentée avec `pi` (cf. `?Constants`)

Si vous appliquer une fonction à une valeur qui ne fait pas partie du domaine de définition de la fonction, le résultat sera une valeur manquante notée `NaN` (*not a number*). Un message d'avertissement va apparaître si vous appliquez une fonction en dehors de son domaine de définition.

Les notions de + l'infini et - l'infini sont notées `Inf` et `-Inf`.

```
## logarithmes et exponentielles
log(1)
log10(100)
log(100, base = 10)
exp(1)

## racine carrée
sqrt(x_2^2)

## valeur absolue
abs(10)
abs(-10)

## fonctions trigonométriques
cos(1)
sin(1)
tan(1)
pi
2 * pi * 10 # circonférence d'un cercle de rayon 10

## si on utilise une valeur en dehors du domaine d'application de la fonction
log(-1) # NaN, car -1 est en dehors du domaine de définition de la fonction log
sqrt(-2) # -2 est en dehors du domaine de définition de la fonction racine carrée

## notions de + ou - l'infini
1 / 0 # [1] Inf
-1 / 0 # [1] -Inf
```

2.4.2.3 Fonctions d'arrondi ♠

Plusieurs fonctions sont disponibles dans R pour arrondir une valeur (cf. `?Round`) :

- la fonction `round()` est utile pour arrondir les décimales. Il faut préciser en argument le nombre de chiffres après la virgule. **Attention** : si le nombre se termine par un 5, l'arrondi se fait vers le chiffre pair le plus proche : 4,45 s'arrondit à 4,4 (la valeur arrondie inférieure) et 4,75 s'arrondit à 4,8 (la valeur arrondie supérieure)
- la fonction `signif()` arrondit aux chiffres les plus significatifs (les plus grands)

- la fonction `floor()` arrondit la valeur à l'entier inférieur
- la fonction `ceiling()` arrondit la valeur à l'entier supérieur
- la fonction `trunc()` ne garde que les entiers, sans arrondir

```
## fonction round()
# l'argument digits permet de définir le nombre de chiffres après la virgule
# exemple si vous voulez arrondir à 2 chiffres après la virgule
round(0.09400, digits = 2) # 0.09
round(0.08600, digits = 2) # 0.09
# arroudir à 1 chiffre après la virgule
round(4.450, digits = 1) # 4.4 ; arrondit au chiffre pari plus proche
round(4.750, digits = 1) # 4.8 ; arrondit au chiffre pari la plus proche
# pour arrondir une valeur 5, le résultat va vers le chiffre pair le plus proche
round(4.5, digits = 0) # 4 ; arrondit au chiffre pair le plus proche
round(1.5, digits = 0) # 2 ; arrondit au chiffre pair le plus proche

## fonction signif()
# on garde les valeurs les plus significative, définie par l'argument digits
signif(123.456789, digits = 1) # 100
signif(123.456789, digits = 2) # 120
signif(123.456789, digits = 3) # 123
signif(123.456789, digits = 4) # 123.5
signif(123.456789, digits = 5) # 123.46 arrondit au chiffre pair le plus proche
signif(4.45, digits = 3) # 4.45
signif(4.45, digits = 2) # 4.4 ; arrondit au chiffre pair le plus proche
signif(4.75, digits = 3) # 4.75
signif(4.75, digits = 2) # 4.8 ; arrondit au chiffre pair le plus proche

## fonction trunc() supprime simplement les décimales
# note : ici, il n'y a pas d'arrondi vers la chiffre pair la plus proche
trunc(123.456) # 123
trunc(4.5) # 4
trunc(1.5) # 1

## la fonction floor() arrondit à l'entier inférieur
floor(4.1) # 4
floor(4.9) # 4

## la fonction ceiling() arrondit à l'entier supérieur
ceiling(4.1) # 5
ceiling(4.9) # 5
```

2.4.3 Concaténation de chaînes de caractères

On peut concatener deux objets en chaînes de caractères :

- la fonction `paste()` concatène les chaînes de caractères en séparant les valeurs par un espace (argument par défaut, cf `?paste`). Cet argument peut être modifié.
- la fonction `paste0()` concatène les chaînes de caractères sans espace.

```
x1 <- "Bonjour"
x2 <- "Toulouse"
paste(x1, x2)
```

```
## [1] "Bonjour Toulouse"
```

```r
paste0(x1, x2)
```

```
## [1] "BonjourToulouse"
```

```r
paste(x1, x2, sep = ", ") # ici on sépare x1 et x2 par une virgule et un espace
```

```
## [1] "Bonjour, Toulouse"
```

```r
# vous pouvez inclure des nombres qui seront transformés en caractères
paste0(x1, 123, x2)
```

```
## [1] "Bonjour123Toulouse"
```

2.4.4 Valeurs logiques `TRUE` et `FALSE`

2.4.4.1 Evaluer des conditions

Nous pouvons utiliser les opérateurs de comparaison ci-dessous (utiles pour évaluer des conditions) :

- `==` ... est égal à ...
- `!=` ... est différent de ...
- `<` ... est inférieur à ...
- `>` ... est supérieur à ...
- `<=` ... est inférieur ou égal à ...
- `>=` ... est supérieur ou égal à ...
- `%in%` ... est inclus dans ...

Par exemple, nous pouvons évaluer les comparaisons suivantes, la réponse attendue est vraie (`TRUE`) ou fausse (`FALSE`).

```r
5 == 10 # est-ce que 5 est égal à 10 ?
5 != 10 # est-ce que 5 est différent de 10 ?
5 < 10 # est-ce que 5 est inférieur à 10 ?
5 > 10 # est-ce que 5 est supérieur à 10 ?
5 <= 5 # est-ce que 5 est inférieur ou égal à 5 ?
5 >= 5 # est-ce que 5 est supérieur ou égal à 5 ?
5 %in% c(4,5,6) # est-ce que 5 est inclus dans le vecteur (4,5,6) ?
5 %in% c(7,8,9) # est-ce que 5 est inclus dans le vecteur (7,8,9) ?
```

La fonction `identical` permet d'évaluer si deux objets sont exactement égaux. Elle peut s'appliquer à des valeurs simples mais aussi à des objets de plus grandes dimensions (vecteurs, matrices, bases de données, ...)

```r
identical(5, 10) # équivalent à la commande 5 == 10
identical(c(1,2,3), c(1,2,3)) # les deux vecteurs (1,2,3) sont bien les mêmes
```

Une comparaison à une valeur manquante (`NA`) retournera une valeur manquante.

Attention, si vous souhaitez évaluer si une valeur est manquante, il faut utiliser la fonction `is.na(x)` (plutôt que `x == NA` qui est déconseillé).

```
NA < 10 # retourne une valeur manquante (pas de solution à cette condition)

is.na(10) # éviter d'utiliser 10 == NA pour tester si une valeur est manquante
is.na(NA)
is.na(c(1,2,3,NA,5,6,NA,8,9,10))
```

2.4.5 Opérations sur des valeurs logiques

On peut combiner des valeurs logiques avec les opérateurs logiques ET, OU, et NON (négation logique)

- `&` opérateur ET
- `|` opérateur OU (sur windows, combinaison de touches altgr + 6 ; sur macOS, combinaison de touche alt + maj + L)
- `!` opérateur NON (négation logique : "n'est pas")

Les résultats attendus d'une combinaison d'opérateurs logiques sont résumés dans les table de vérité ci-dessous.

- Opérateur ET

a	b	a ET b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

- Opérateur OU

a	b	a OU b
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

- Opérateur NON

a	NON a
TRUE	FALSE
FALSE	TRUE

```
## opérateur ET
TRUE & TRUE
TRUE & FALSE
FALSE & TRUE
FALSE & FALSE
(5 > 10) & (2 != 5) # TRUE ET TRUE donne TRUE
(5 > 10) & (2 == 5) # TRUE ET FALSE donne FALSE
(5 < 10) & (2 == 5) # FALSE ET FALSE donne FALSE
```

```
## opérateur OU
TRUE | TRUE
TRUE | FALSE
FALSE | TRUE
FALSE | FALSE
(5 < 10) | (2 != 5) # TRUE OU TRUE donne TRUE
(5 > 10) | (2 != 5) # FALSE OU TRUE donne TRUE
(5 > 10) | (2 == 5) # FALSE OU FALSE donne FALSE

## opérateur NON
!TRUE
!FALSE
!(5 < 10) # non-TRUE donne FALSE
!(5 > 10) # non-FALSE donne TRUE
```

Une opération logique avec une valeur manquante retournera une valeur manquante, sauf si l'information manquante n'est pas bloquante pour l'opération.

```
## opérateur NON
!NA # retourne une valeur manquante

## opérateur ET
TRUE & NA # retourne une valeur manquante
FALSE & NA # retourne une valeur FALSE (car la réponse est fausse, quelle que
           # soit la valeur qui aurait été à la place de NA)

## opérateur OU
FALSE | NA # retourne une valeur manquante
TRUE | NA # retourne une valeur TRUE (car la réponse est vraie, quelle que
           # soit la valeur qui aurait été à la place de NA)
```

Il existe également un opérateur `xor()` correspondant au OU EXCLUSIF (mais il semble peu utilisé en pratique) :

- Opérateur OU EXCLUSIF

a	b	a OU EXCLUSIF b
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

```
# opérateur OU EXCLUSIF
xor(TRUE, TRUE)
xor(TRUE, FALSE)
xor(FALSE, TRUE)
xor(FALSE, FALSE)
xor((5 < 10), (2 != 5)) # (TRUE) OU exclusif (TRUE) donne FALSE
xor((5 > 10), (2 != 5)) # (FALSE) OU exclusif (TRUE) donne TRUE
xor((2 != 5), (5 > 10)) # (TRUE) OU exclusif (FALSE) donne TRUE
xor((5 > 10), (2 == 5)) # (FALSE) OU exclusif (FALSE) donne FALSE
```

2.5 Les vecteurs atomiques `c()`

2.5.1 Création d'un vecteur atomique

La fonction `c()` (pour concaténer) permet de créer des vecteurs atomiques. Les valeurs d'un vecteur atomique doivent toutes être du même type et séparées par une virgule.

```
## Un vecteur de 3 nombres réels :
vect_dbl <- c(1, pi, 3.54)
print(vect_dbl) # [1] 1.000000 3.141593 3.540000

## Un vecteur de 5 entiers :
vect_int <- c(5L, 4L, -3L, 15L, -8L)
print(vect_int) # [1]  5  4 -3 15 -8

## Un vecteur de 4 chaînes de caractères :
vect_char <- c("a", "B", "XYZ", "HELLO")
vect_char # [1] "a"     "B"     "XYZ"   "HELLO"

## Un vecteur logique :
## pour les valeurs logiques, on peut utiliser les abbréviations T pour TRUE,
## et F pour FALSE
vect_logic <- c(TRUE, FALSE, FALSE, F, T, T)
print(vect_logic) # [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

Vous pouvez également combiner plusieurs vecteurs pour créer un vecteur plus long en indiquant plusieurs vecteurs au sein de la fonction `c()`.

```
vect1 <- c(1, 2, 3)
vect2 <- c(4, 5, 6)
vect3 <- c(7, 8, 9)
vect_combine <- c(vect1, vect2, vect3)
vect_combine
# [1] 1 2 3 4 5 6 7 8 9
```

2.5.2 Créer une séquence de valeurs

Pour créer une séquence de nombre entiers, ascendante ou descendante de 1 en 1, vous pouvez utiliser la commande :

```
vect_1a10 <- 1:10 # vecteur ascendant
vect_1a10 # [1]  1  2  3  4  5  6  7  8  9 10

vect_3_a_moins5 <- 3:-5 # vecteur descendant
vect_3_a_moins5 # [1] 3  2  1  0 -1 -2 -3 -4 -5
```

La commande `seq()` permet de définir des séquences de manière plus flexible, grâce aux arguments `from`, `to`, `by`, `length.out`, `along.with` (regardez l'aide `?seq()`).

```
seq(from = 1, to = 10) # vecteur de 1 à 10, équivalent à la commande 1:10
seq(from = 0, to = 10, by = 2) # vecteur de 0 à 10, de 2 en 2

seq(from = 1, to = 50, length.out = 5) # vecteur de 1 à 50, de longueur 5
# les intervalles entre les valeurs sont calculés automatiquement par la formule
```

```r
# (valeur de départ - valeur d'arrivée) / (longueur totale - 1)

seq(from = 0, to = 60, along.with = c("dix", "vingt", "trente"))
# vecteur de 0 à 60, dont la longueur est égale à la longeur du vecteur
# c("dix","vingt","trente") (sa longeur est de 3)

# avec l'argument 'by', si le cycle ne tombe pas juste, la séquence s'arrête
# avant la dernière valeur indiquée par 'to = '
seq(from = 1, to = 10, by = 2)
# [1] 1 3 5 7 9
# la séquence ne va pas jusqu'à 10, elle s'arrête à 9, car 9 + 2 = 11
# ce qui dépasserait la valeur maximale demandée par 'to = 10'
```

On peut combiner des vecteurs de chaînes de caractères à des vecteurs numériques :

```r
paste0("L", seq(from = 1, to = 10))
# on obtient le vecteur :
# "L1"  "L2"  "L3"  "L4"  "L5"  "L6"  "L7"  "L8"  "L9"  "L10"
# dans cet exemple, le vecteur à une seule valeur c("L") est "recyclé" pour être
# concaténé à chacun des éléments du vecteur c(1,2,3,4,5,6,7,8,9,10)

paste0(c("A", "B", "C"), seq(from = 1, to = 10))
# on obtient le vecteur :
# "A1"  "B2"  "C3"  "A4"  "B5"  "C6"  "A7"  "B8"  "C9"  "A10"
# ici, le vecteur c("A", "B", "C") est "recyclé" pour être concaténé à chaque
# élément du vecteur seq(1,10)
```

2.5.3 Créer un vecteur de valeurs répétées

La fonction `rep` permet de créer des vecteurs de valeurs répétées. Les arguments `times`, `each`, `length.out` permettent de préciser comment les valeurs doivent être répétées (voir dans l'aide `?rep`)

```r
rep(5, times = 3) # répète la valeur 5, 3 fois.
# [1] 5 5 5

rep(1:4, times = 3) # repète 3 fois le vecteur c(1, 2, 3, 4)
# [1] 1 2 3 4 1 2 3 4 1 2 3 4

rep(1:4, each = 3) # chaque élément du vecteur c(1, 2, 3, 4) est répété 3 fois
# [1] 1 1 1 2 2 2 3 3 3 4 4 4

rep(1:4, length.out = 10) # le vecteur c(1,2,3,4) est répété dans un vecteur
                          # dont la longueur totale est de 10
# [1] 1 2 3 4 1 2 3 4 1 2
```

2.5.4 Opérations arithmétiques sur un vecteur

On appliquer une opération arithmétique avec un scalaire (une valeur simple) à chacun des éléments d'un vecteur :

```r
## additionne +2 à chaque élément de la séquence c(1,2,3,4,5)
seq(from = 1, to = 5) + 2 # cela donne 3 4 5 6 7

## soustrait -2 à chaque élément de la séquence c(1,2,3,4,5)
```

```r
seq(from = 1, to = 5) - 2 # cela donne -1  0  1  2  3

## multiplie par à chaque élément de la séquence c(1,2,3,4,5)
seq(from = 1, to = 5) * 5 # cela donne 5 10 15 20 25

## divise par 5 chaque élément de la séquence c(1,2,3,4,5)
seq(from = 1, to = 5) / 5 # cela donne 0.2 0.4 0.6 0.8 1.0
```

Si vous appliquez une opération arithmétique entre deux vecteurs de même longueur, l'opération se fera entre les 1ers éléments de chaque vecteur, puis entre les 2èmes éléments de chaque vecteur, etc.

```r
vec_A <- c(1, 2, 3, 4, 5)
vec_B <- c(-1, +1, -2, +2, -5)

# addition : c((1 + (-1)), (2 + 1), (3 + (-2)), (4 + 2), (5 + (-5)))
vec_A + vec_B # 0 3 1 6 0

# soustraction : c((1 - (-1)), (2 - 1), (3 - (-2)), (4 - 2), (5 - (-5)))
vec_A - vec_B # 2  1  5  2 10

# multiplication : c((1 * (-1)), (2 * 1), (3 * (-2)), (4 * 2), (5 * (-5)))
vec_A * vec_B  # -1   2  -6   8 -25

# division : c((1 / (-1)), (2 / 1), (3 / (-2)), (4 / 2), (5 / (-5)))
vec_A / vec_B # -1.0  2.0 -1.5  2.0 -1.0
```

Si vous appliquez une opération arithmétique entre deux vecteurs de longueur différente, l'opération se fera entre les 1ers éléments de chaque vecteur, puis entre les 2èmes éléments de chaque vecteur, etc. Lorsqu'on arrive au bout du vecteur le plus court, les opérations continuent en reprenant à partir de la première valeur du vecteur le plus court (le vecteur est "recyclé"). Un message d'avertissement vous prévient également lorsque la longueur du vecteur le plus long n'est pas un multiple de la longueur du vecteur le plus court (mais cela n'empêche pas l'opération de se faire).

```r
vec_A <- c(1, 2, 3, 4, 5)
vec_C <- c(1, 2, 3)

vec_A + vec_C # 2 4 6 5 7
# addition : c((1 + 1), (2 + 2), (3 + 3), (4 + 1), (5 + 2))

vec_A - vec_C # 0 0 0 3 3
# soustraction : c((1 - 1), (2 - 2), (3 - 3), (4 - 1), (5 - 2))

vec_A * vec_C  # 1  4  9  4 10
# multiplication : c((1 * 1), (2 * 2), (3 * 3), (4 * 1), (5 * 2))

vec_A / vec_C # 1.0 1.0 1.0 4.0 2.5
# division : c((1 / 1), (2 / 2), (3 / 3), (4 / 1), (5 / 2))
```

2.5.5 Opérations logiques sur un vecteur

On peut faire des opérations logiques sur des vecteurs (voir les tables de vérité plus haut) :

```r
### 1) Est-ce que les éléments de c(1,2,3) sont inclus dans c(1,3,5,7,9) ?
c(1, 2, 3) %in% seq(from = 1, to = 9, by = 2)
# TRUE FALSE  TRUE
# les valeurs 1 et 3 sont bien comprise dans le vecteur c(1, 3, 5, 7, 9),
# mais pas la valeur 2

### 2) Est-ce que les éléments de c(1,3,5,7,9) sont inclus dans c(1,2,3) ?
seq(from = 1, to = 9, by = 2) %in% c(1, 2, 3)
# TRUE  TRUE FALSE FALSE FALSE
# les valeurs 1 et 3 sont bien comprise dans le vecteur c(1, 2, 3),
# mais pas les valeurs 5, 7 et 9

### 3) opération ET entre les éléments de deux vecteurs logiques
c(TRUE, TRUE, FALSE, FALSE) & c(TRUE, FALSE, TRUE, FALSE)
# TRUE FALSE FALSE FALSE
# 1er élément avec le 1er élément     TRUE & TRUE = TRUE
# 2ème élément avec le 2ème élément  TRUE & FALSE = FALSE
# 3ème élément avec le 3ème élément  FALSE & TRUE = FALSE
# 4ème élément avec le 4ème élément FALSE & FALSE = FALSE

### 4) opération OU entre les éléments de deux vecteurs logiques
c(TRUE, TRUE, FALSE, FALSE) | c(TRUE, FALSE, TRUE, FALSE)
# TRUE  TRUE  TRUE FALSE
# 1er élément avec le 1er élément     TRUE | TRUE = TRUE
# 2ème élément avec le 2ème élément  TRUE | FALSE = TRUE
# 3ème élément avec le 3ème élément  FALSE | TRUE = TRUE
# 4ème élément avec le 4ème élément FALSE | FALSE = FALSE
```

2.5.5.1 Quelques fonctions logiques utiles

Les fonctions suivantes sont souvent utilisées pour réaliser des opérations logiques :

- la fonction `which()` indique la "position" des réponses vraies dans un vecteur de tests logiques
- la fonction `any()` retourne un résultat `TRUE` si la condition entre parenthèse est vraie pour au moins une des valeurs du vecteur, sinon le résultat est `FALSE`
- la fonction `all()` retourne un résultat `TRUE` si la condition entre parenthèse est vraie pour toutes les valeurs du vecteur, sinon le résultat est `FALSE`

```r
## Commençons avec ce vecteur de réels
vect_test <- c(1.5, 5.83, 3.2, 15, 9.99)

## pour connaître la position des valeurs supérieures à 5 dans ce vecteur :
which(vect_test > 5)
# [1] 2 4 5
# la 2ème valeur (5.83), la 4ème valeur (15) et la 5ème valeur (9.99)
# dans le vecteur sont supérieures à 5

## ce sont les positions des réponses 'TRUE' à l'opération logique :
vect_test > 5
# [1] FALSE  TRUE FALSE  TRUE  TRUE

## Est-ce qu'une de ces valeurs est inférieure à 3 ?
any(vect_test < 3) # TRUE

## Est-ce qu'une de ces valeurs est inférieure à 1 ?
```

```r
any(vect_test < 1) # FALSE

## Est-ce que l'ensemble de ces valeurs est inférieure à 3 ?
all(vect_test < 3) #  FALSE

## Est-ce que l'ensemble de ces valeurs sont supérieures à 1 ?
all(vect_test > 1) # TRUE
```

2.5.6 Principe de coercition

Commes les valeurs d'un vecteur atomique doivent toutes être du même type, si vous combinez des vecteurs dont les valeurs sont de types différents, R va transformer le résultat dans un seul type de valeur, par **coercition** (*coercion*). Les règles de coercition sont les suivantes :

- combiner un vecteur `character` avec d'autres types (`double`, `integer` ou `logical`) résulte en un vecteur `character` (les valeurs sont toutes transformées en format caractère `"X"`)
- combiner un vecteur `double` (réel) avec un `integer` ou `logical` résulte en un vecteur `double` (les entiers sont transformés en réels, et les valeurs logiques `TRUE` deviennent `1` et `FALSE` deviennent `0`, en format de réels)
- combiner un vecteur `integer` (entier) avec un `logical` résulte en un vecteur `integer` (les valeurs logiques `TRUE` deviennent `1L`, et les valeurs `FALSE` deviennent `0L`, en format d'entiers).

```r
vect_dbl <- c(1, pi, 3.54) # un vecteur de réels
vect_int <- c(5L, 4L, -3L, 15L, -8L) # un vecteur d'entiers
vect_char <- c("a", "B", "XYZ", "HELLO") # un vecteur de caractères
vect_logic <- c(TRUE, FALSE, FALSE, F, T, T) # un vecteur logique

## On vérifie ce principe de coercition
vect1 <- c(vect_dbl, vect_int, vect_char, vect_logic)
vect1
typeof(vect1) # "character"

vect2 <- c(vect_dbl, vect_int, vect_logic)
vect2
typeof(vect2) # "double"

vect3 <- c(vect_int, vect_logic)
vect3
typeof(vect3) # "integer"
```

2.5.7 Attributs d'un vecteur

Un vecteur a deux principaux attributs :

- sa **classe**, que l'on peut obtenir avec la fonction `class()`
- sa **longueur**, c'est-à-dire le nombre de valeurs qu'il contient, que l'on peut obtenir avec la fonction `length()`

Pour décrire de manière résumé les attributs d'un vecteur, vous pouvez utiliser la fonction `str()` : cette fonction vous indiquera la classe du vecteur, sa longueur et affichera également les premières valeurs. C'est la fonction qui est appliquée pour décrire les objets dans la fenêtre "environnement" de RStudio.

```r
## on reprend les vecteurs vus précédemment :
vect_dbl <- c(1, pi, 3.54)
vect_int <- c(5L, 4L, -3L, 15L, -8L)
```

```r
vect_char <- c("a", "B", "XYZ", "HELLO")
vect_logic <- c(TRUE, FALSE, FALSE, F, T, T)

## la fonction class() permet de décrire le type du vecteur
class(vect_dbl) # numeric     Note : ici, R indique le mode "numeric" plutôt que
                #                    le type "double"
class(vect_int) # integer
class(vect_char) # character
class(vect_logic) # logical

## la fonction length() permet de décrire la longueur du vecteur (c'est à dire
## le nombre d'éléments qu'il contient)
length(vect_dbl) # 3
length(vect_int) # 5
length(vect_char) # 4
length(vect_logic) # 6

## description sommaire des attributs des vecteurs
str(vect_dbl) # num [1:3] 1 3.14 3.54
str(vect_int) # num [1:3] 1 3.14 3.54
              # note : la fonction str() indique le mode "numeric" plutôt que
              #        que le type "integer"
str(vect_char) # chr [1:4] "a" "B" "XYZ" "HELLO"
str(vect_logic) # logi [1:6] TRUE FALSE FALSE FALSE TRUE TRUE
```

Il est possible d'ajouter des attributs à un vecteur. Les attributs peuvent être considérés comme des méta-données associées au vecteur.

Un des attributs les plus fréquents est de nommer chaque élément d'un vecteur avec la fonction `names()` : on associe un vecteur de noms (en caractères) aux valeurs du vecteur.

```r
## par exemple, si on crée le vecteur de réels suivant
sex <- c(1, 1, 2, 1, 2, 2)
names(sex) <- c("homme", "homme", "femme", "homme", "femme", "femme")
sex
# homme homme femme homme femme femme     le nom apparaît au dessus des valeurs
#     1     1     2     1     2     2

## mais on peut nommer les valeurs de façons parfaitement arbitraire :
vect_bizarre <- c(1, 2, 3, 4, 5)
names(vect_bizarre) <- c("un", "trois", "douze", "douze", "2")
vect_bizarre
# un trois douze douze     2             le nom apparaît au dessus des valeurs
#  1     2     3     4     5
```

La fonction `attr()` permet d'obtenir ou de définir un attribut spécifique (pour compléter les méta-données). La fonction `attributes()` indique l'ensemble des attributs associés à un vecteur.

```r
## La fonction attr() permet de récupérer des attributs spécifiques
attr(sex, "names")
# [1] "homme" "homme" "femme" "homme" "femme" "femme"

## Cette fonction permet également de définir de nouveux attributs
## Ci-dessous, on définit un nouvel attribut "var_name" auquel on associe
## la valeur "Sexe du participant" pour stocker la méta-donnée indiquant le
## nom complet de la variable.
```

```r
attr(sex, "var_name") <- c("Sexe du participant")
sex
# homme homme femme homme femme femme
#     1     1     2     1     2     2
# attr(,"var_name")
# [1] "Sexe du participant"

## La fonction attributes() permet de décrire l'ensemble des attributs du vecteur
attributes(sex)
# $names
# [1] "homme" "homme" "femme" "homme" "femme" "femme"
#
# $var_name
# [1] "Sexe du participant"

## On peut récupérer également chaque attribut avec l'opérateur dollar $
attributes(sex)$names
# [1] "homme" "homme" "femme" "homme" "femme" "femme"
attributes(sex)$var_name
# [1] "Sexe du participant"
```

2.5.8 Vecteurs de type `factor`

Les vecteurs de types `factor` sont utiles pour ajouter certaines contraintes propres aux variables qualitatives. Ce sont des vecteurs qui contiennent uniquement des valeurs prédéfinies (connues dès le protocole de l'expérience). Par exemple, en amont de l'expérience, on peut avoir défini que le niveau d'étude se mesurera avec 3 modalités : "lycée", "bac", et "université".

Cette caractérisation sera utile pour utiliser la variable dans des contextes précis, par exemple :

- dans un modèle de régression (où on veut que le logiciel créé automatiquement des indicatrices pour prendre en compte cette variable qualitative de manière adéquate),
- pour représenter graphiquement la variable (par exemple, identifier automatiquement que ces valeurs doivent être décrites avec un diagramme en barres plutôt qu'en box-plot).
- dans des analyes descriptives, cela permettra d'identifier directement si certaines modalités de réponses n'apparaîssent pas au sein des valeurs observées (le décompte pour cette modalité de réponse sera égal à 0).

Les vecteurs de type `factor` sont construits par dessus des vecteurs d'entiers, avec 2 attributs :

- un attribut `class`, qui indique "factor" et permet d'identifier ce vecteur en tant que `factor`.
- un attribut `levels`, qui définit les valeurs possibles (définies a priori). Dans notre exemple, ce sont les 3 valeurs "lycée","bac", et "université".

Pour créer un vecteur de type factor, on utilise la fonction `factor()`. Par défaut, R va identifier les 3 valeurs uniques présentes dans le vecteur, puis les ranger de la plus petite à la plus grande (selon la valeur numérique ou l'ordre alphabétique) pour créer l'attribut `levels`. L'attribut levels est un vecteur de caractères.

```r
## Exemple avec une variable mesurant le niveau d'étude :
fact_1 <- factor(c("univ", "bac", "bac", "lycée", "univ", "lycée", "bac"))
fact_1
# [1] univ  bac   bac   lycée univ  lycée bac
# Levels: bac lycée univ
```

```r
## on peut récupérer les 2 attributs de ce 'factor' avec la fonction attributes()
attributes(fact_1)
# $levels
# [1] "bac"   "lycée" "univ"
# $class
# [1] "factor"

## l'attribut 'levels' est un vecteur de caractères, rangé par ordre alphabétique
attributes(fact_1)$levels
# [1] "bac"   "lycée" "univ"

## Un entier est associé à chaque élément de l'attribut 'levels'  :
##  - "bac" est associé à 1,
##  - "lycée" est associé à 2,
##  - "univ" est associé à 3
str(fact_1)
# Factor w/ 3 levels "bac","lycée",..: 3 1 1 2 3 2 1

## si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_1)
# [1] 3 1 1 2 3 2 1
```

On aurait sans doute préféré que l'ordre des `levels` commence avec le "lycée" et termine avec "univ". Pour cela il est possible de définir nous même le vecteur de levels avec l'argument `levels` de la fonction `factor` :

```r
## On utilise l'argument 'levels' directement dans la fonction 'factor' :
fact_1bis <- factor(c("univ", "bac", "bac", "lycée", "univ", "lycée", "bac"),
                    levels = c("lycée", "bac", "univ"))
fact_1bis
# [1] univ  bac   bac   lycée univ  lycée bac
# Levels: lycée bac univ

# cette fois ci, les levels sont dans l'ordre qui nous convient.

## Un entier est associé à chaque élément de l'attribut 'levels', en suivant
## l'ordre de ses éléments
##  - "lycée" est associé à 1,
##  - "bac" est associé à 2,
##  - "univ" est associé à 3

## si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_1bis)
# [1] 3 2 2 1 3 1 2
```

Si la variable qualitative a été codée avec un codage numérique, on peut également appliquer la fonction `factor()`. Prenons l'exemple de la variable "tendance" codée : -1 pour une diminution, 0 pour une stabilité, et 1 pour une augmentation.

```r
## Exemple avec une variable mesurant une tendance sous force de codage numérique :
fact_2 <- factor(c(0, 1, 0, -1, 0, 0, -1, 1, -1, 0, 1, 1))
fact_2
#  [1] 0  1  0  -1 0  0  -1 1  -1 0  1  1
# Levels: -1 0 1

str(fact_2)
# Factor w/ 3 levels "-1","0","1": 2 3 2 1 2 2 1 3 1 2 ...
```

```
## Après transformation du vecteur en 'factor', les valeurs -1, 0 et 1 vont
## apparaître sous format de caractères "-1", "0" et "1".

## R a identifié automatiquement les 3 valeurs uniques : -1, 0, et 1,
## les a rangé de la plus petite à la plus grande.
attributes(fact_2)
# $levels
# [1] "-1" "0"  "1"
# $class
# [1] "factor"

## Un entier est associé à chaque élément de l'attribut 'levels'  :
## "-1" est associé à 1,
## "0" est associé à 2,
## "1" est associé à 3

## si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_2)
# [1] 2 3 2 1 2 2 1 3 1 2 3 3
```

On peut noter que les entiers sur lesquels les vecteurs `factor` sont construits commencent toujours à 1 et augmente de 1 en 1. Il n'est pas possible de faire autrement (par exemple d'associer une valeur 0 à une des modalités de réponses).

On voit que les vecteurs de type `factor` ne sont pas pratiques pour manipuler de manière flexible le codage de variables qualitatives (attribuer les codes et les étiquettes de manière flexible). Pour une meilleure gestion des méta-données (des noms de variables, des codages et des étiquettes associées aux modalités de réponses), il est préférable de créer une base de données des méta-données. Nous verrons comment faire au chapitre 3.

Les variables qualitatives peuvent également être définies comme des **facteurs ordonnés** avec la fonction `ordered`. Cela peut être utile pour des variables à utiliser dans le cadre de régression multinomiales par exemple. Le comportement d'un vecteur de type `ordered` est très proche de celui d'un `factor` (le type `ordered` est une petite variation du type `factor`).

```
## La variable de niveau d'étude est une variable qualitative ordinale,
## on peut la définir également en tant que vecteur 'ordered'
fact_1ter <- ordered(c("univ", "bac", "bac", "lycée", "univ", "lycée", "bac"),
                     levels = c("lycée", "bac", "univ"))
fact_1ter
# [1] univ  bac   bac   lycée univ  lycée bac
# Levels: lycée < bac < univ
## on voit que R considère que "univ" est supérieur à "bac", lui même supérieur
## à "lycée"

attributes(fact_1ter)
# $levels
# [1] "lycée" "bac"   "univ"
#
# $class
# [1] "ordered" "factor"
## le vecteur est à la fois de type "factor" et de type "ordered"

## Un entier est associé à chaque élément de l'attribut 'levels', en suivant
## l'ordre de ses éléments
##  - "lycée" est associé à 1,
##  - "bac" est associé à 2,
##  - "univ" est associé à 3
```

```r
## si on force par coercition à afficher le vecteur au format d'entiers :
as.integer(fact_1ter)
# [1] 3 2 2 1 3 1 2
```

2.5.9 Vecteurs de type Date ou Date-time

Les vecteurs de type `Date` ou `Date-time` sont construits à partir de valeurs réelles, avec un attribut permettant de les prendre en compte comme des "dates" ou des "dates-heures".

Les **vecteurs de types** `Date` ont un attribut de classe égale à `"Date"`. La valeur réelle sous jacente correspond au nombre de jours depuis le 1er janvier 1970. Le vecteur affiche ses valeurs au format "aaaa-mm-jj" pour indiquer l'année, le mois et le jour, séparés d'un tiret et entre guillemets. On peut saisir des valeurs de dates avec un vecteur de caractères au sein de la fonction `as.Date()`

```r
## on saisie un vecteur de 2 dates :
vect_dt <- as.Date(c("1970-01-31", "1971-01-01"))
vect_dt

typeof(vect_dt)
# [1] "double"      les valeurs sous-jacente sont des réels

attributes(vect_dt)
# $class
# [1] "Date"

## si on force le vecteur vect_dt à donner les valeurs au format de réels,
## on retrouve le nombre de jours depuis le 1er janvier 1970.
## (respectivement, 30 jours et 365 jours dans cet exemple)
as.double(vect_dt)
# [1]  30 365
```

Les **vecteurs de types "Date-time"** indiquent une date et une heure. La valeur réelle sous jacente correspond au nombre de secondes depuis le 1er janvier 1970. Ces vecteurs ont deux attributs :

- un attribut de classe égale à `POSIXct` pour *Portable Operating System Interface - calender time* (ou encore `POSIXlt`, pour *Portable Operating System - local time*, mais ce format semble moins pratique pour une base de données).
- un attribut de fuseau horaire `tzone` (pour *time zone*). Cet attribut aura simplement un effet sur l'heure affichée, la valeur réelle sous-jacente reste la même.

Le vecteur affiche ses valeurs au format "aaaa-mm-jj hh:mm" pour indiquer l'année, le mois, le jour, l'heure et les minutes, entre guillemets. On peut saisir des valeurs de dates avec un vecteurs de caractères au sein de la fonction `as.POSIXct()`

```r
## on saisie un vecteur de 2 dates-heures :
vect_dt_time <- as.POSIXct(c("1970-01-01 01:30", "1971-01-02 00:00"),
                           tz = "UTC") # pour le fuseau du méridien de Greenwich
vect_dt_time # [1] "1970-01-01 01:30:00 UTC" "1971-01-02 00:00:00 UTC"

typeof(vect_dt_time)
# [1] "double"      les valeurs sous-jacentes sont des réels

attributes(vect_dt_time)
# $class
```

```
# [1] "POSIXct" "POSIXt"
#
# $tzone
# [1] "UTC"

## si on force le vecteur vect_dt_time à donner les valeurs au format de réels,
## on retrouve le nombre de secondes depuis le 1er janvier 1970 à minuit (00:00).
as.double(vect_dt_time)
# [1]    5400 31622400

# Entre le 1er janvier 1970 à minuit et le 1er janvier 1970 à 1h30, il s'est
# écoulé 60 * 90 secondes = 5400 secondes.

## Si vous souhaitez appliquer le fuseau horaire de Paris à ces valeurs :
attr(vect_dt_time, "tzone") <- "Europe/Paris"
vect_dt_time
# [1] "1970-01-01 02:30:00 CET" "1971-01-02 01:00:00 CET"
## la valeur s'affiche avec 1 heure de décalage (par rapport à Londres)
```

2.5.10 Sélection par indexation

Pour sélectionner des sous-ensembles de valeurs au sein d'un vecteur, une des méthodes très utilisée dans R est **l'indexation**, en ajoutant des crochets `[]` à la fin d'un vecteur. L'indexation se fait principalement selon 3 approches :

- en indiquant la **position** des valeurs que l'on veut sélectionner
- en indiquant le **nom** des valeurs que l'on veut sélectionner (définis selon l'attribut `name` du vecteur)
- en indiquant une **condition**, seules les valeurs dont la condition est "vraies" (`TRUE`) seront sélectionnées

2.5.10.1 Indexation sur la position

On peut sélectionner un sous-ensemble d'un vecteur en indiquant entre crochets `[]` la position des valeurs que l'on souhaite sélectionner. Il est également possible d'exclure des valeurs en indiquant un signe `-` devant la position de la valeur.

Si vous indiquez une (ou plusieurs) positions plus grande(s) que la longueur du vecteur, on obtient des données manquantes.

```
## On reprend le vecteur de réels suivant :
vect_test <- c(1.5, 5.83, 3.2, 15, 9.99)

## Pour sélectionner la 4ème valeur (15), on peut utiliser la notation []
vect_test[4]
# [1] 15

## Pour sélectionner la 1ère, la 2ème et la 5ème valeur :
vect_test[c(1, 2, 5)]
# [1] 1.50 5.83 9.99

## l'ordre dans lequel on indique la position définie l'ordre dans lequel le
## résultat est donné :
vect_test[c(5, 1, 2)]
# [1] 9.99 1.50 5.83
```

```r
## Pour sélectionner les valeurs en position 3, 4 et 5, on peut utiliser la
## la notation de séquence ":"
vect_test[3:5]
# [1]  3.20 15.00  9.99

## Pour exclure des valeurs de la sélection, on peut utilise le signe "-"
vect_test[-c(1, 2, 5)]
# [1]  3.2 15.0
# il garde la 3ème et la 4ème valeur, après avoir exclu les valeurs aux
# positions 1, 2 et 5

## Si on indique des positions plus grande que la longueur du vecteur,
## on obtient des données manquantes :
vect_test[c(1, 3, 8, 9)]
# [1] 1.5 3.2  NA  NA
```

2.5.10.2 Indexation sur le nom

Il est également possible de sélectionner un sous-ensemble d'un vecteur en indiquant entre crochets `[]` le nom indiqué dans l'attribut `name` du vecteur.

```r
## On reprend les deux vecteurs nommés vus précédemment :
sex <- c(1, 1, 2, 1, 2, 2)
names(sex) <- c("homme", "homme", "femme", "homme", "femme", "femme")
sex
# homme homme femme homme femme femme     le nom apparaît au dessus des valeurs
#     1     1     2     1     2     2

vect_bizarre <- c(1, 2, 3, 4, 5)
names(vect_bizarre) <- c("un", "trois", "douze", "douze", "2")
vect_bizarre
# un trois douze douze     2
#  1     2     3     4     5

## Par exemple en sélectionnant les valeurs de sex nommées "homme", on obtient
sex["homme"]
# homme
#     1

## En sélectionnant les valeurs de vect_bizarres nommées "un" et "trois" :
vect_bizarre[c("un", "trois")]
# un trois
#  1     2

## En sélectionnant les valeurs de vect_bizarres nommées "douze", on obtient
vect_bizarre["douze"]
# douze
#     3

# avec cette méthode d'indexation, on ne peut pas retrouver la deuxième valeur
# associée au nom "douze" (la valeur 4) de vect_bizarre

## Attention, cette méthode d'indexation fonctionne uniquement avec les noms
## donné dans l'attribut name()
## Cela ne fonctionne pas avec les valeurs d'un vecteur caractères :
```

```r
sex_char <- c("homme", "homme", "femme", "homme", "femme", "femme")
sex_char
names(sex_char)
# NULL          l'attribut "names" de ce vecteur caractère ne contient rien

sex_char["homme"]
# [1] NA            # l'indexation n'a rien sélectionné car l'attribut names
                    # ne contient aucun nom
```

En pratique, on voit que lorsque l'attribut `names` contient des noms en doublons, seule la première valeur associée à ces noms en est sélectionnée.

2.5.10.3 Indexation sur une condition

Il est enfin possible de sélection les valeurs à l'aide d'un vecteur logique de même longueur que le vecteur sur lequel on travaille. Les valeurs en position `TRUE` seront sélectionnées, les valeurs en position `FALSE` seront exclues.

On peut se servir de cette approche pour sélectionner les valeurs en appliquant une opération logique.

```r
vect_test <- c(1.50, 5.83, 3.20, 15.00, 9.99)

## On peut indiquer entre crochet les valeurs que l'on veut sélectionner avec
## TRUE dans la position corespondante
vect_test[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
# [1]  1.5  3.2 15.0

## On peut ainsi directement appliquer une condition pour sélectionner les
## valeurs correspondant à cette condition.
## Par exemple, pour sélectionner les valeurs supérieure à 5 :
vect_test[vect_test > 5]
# [1]  5.83 15.00  9.99

## Pour sélectionner les valeurs égale à 5.83 ou à 9.99
vect_test[vect_test == 5.83 | vect_test == 9.99]
# [1] 5.83 9.99

## ou encore
vect_test[vect_test %in% c(5.83, 9.99)]
# [1] 5.83 9.99

## Attention, si vous utilisez en indexation un vecteur logique plus court,
## R va automatiquement recycler le vecteur logique
vect_test[c(TRUE, FALSE, TRUE)]
# [1]  1.5  3.2 15.0

# ici, R a recyclé les valeurs logique en appliquant
# c(TRUE, FALSE, TRUE) au 3 premières valeurs de vect_test, soit 1.5 et 3.2
# puis a recyclé le vacteur logique pour les 3 denières valeurs de vect_test
# en appliquant c(TRUE, FALSE). Le dernier TRUE n'est pas utilisé
# ce qui sélectionne la 4ème valeur 15.0, mais pas la dernière.
```

2.5.10.4 Assignation par indexation

Cette méthode de sélection des éléments d'un vecteur peut être utilisée pour remplacer certaines valeurs du vecteur.

```r
vect_test <- c(1.50, 5.83, 3.20, 15.00, 9.99)

## Par exemple, on veut remplacer les valeurs 5.83 et 9.99 par les valeurs
## 2.2 et une valeur manquante NA

## Ces deux valeurs occupent la position 2 et la position 5 :
vect_test[c(2, 5)] <- c(2, NA)
vect_test
# [1]  1.5  2.0  3.2 15.0   NA
## Les valeurs ont été changées au sein du vecteur
```

2.5.11 Fonctions statistiques pour résumer une série de valeurs

2.5.11.1 Variables quantitatives

Pour décrire les paramètres de la distribution d'une variable quantitative (d'un vecteur de valeurs réelles ou d'entiers), les fonctions suivantes peuvent être utilisées :

- `mean()`, `sd()` et `var()` permettent de calculer respectivement la moyenne, l'écart-type (*standard deviation*) et la variance d'une variable
- `min()`, `max()`, `median()` et `quantile()` permettent de calculer respectivement le minimum, le maximum, la médiane et les quantiles d'une variable. Pour la fonction quantile, il faut préciser en argument les probabilités correspondant aux quantiles demandés, par exemple `probs = c(0, 0.25, 0.5, 0.75, 1)` permet d'obtenir le minimum, le 1er quartile (25ème percentile), la médiane (50ème percentile), le 3ème quartile (75ème percentile) et le maximum.
- `sum()` permet de calculer la somme des valeurs, `prod()` permet de calculer le produit des valeurs

Pour toutes ces fonctions, en présence de données manquantes, le résultat sera également une donnée manquante, à moins d'ajouter l'argument `na.rm = TRUE` (*remove NA*) qui supprime les données manquantes et permet de calculer ces paramètres à partir des valeurs non-manquantes.

Note pour la fonction `quantile`: La méthode de calcul des quantiles est données par l'argument `type`. Par défaut, la méthode appliquée est le "Type 7", utilisée par le logiciel S (ancètre du logiciel R). Le logiciel SAS utilise la méthode de "Type 2", les logiciels Stata, SPSS et Minitab utilisent la méthode de "Type 6". En fonction du type choisi, les résultats peuvent être légèrement différents. cf. `?quantile()`

```r
## on va calculer les paramètres statistique pour la variable quantitative :
var_quanti <- c(4.5, 2.0, 5.5, 10.4, 8.7, NA, 3.2, 4.0, 1.3, NA, 5.7, 1.7)
# on voit qu'elle contient 2 valeurs manquantes

## Si on applique les fonctions mean(), sd(), etc, le résultat sera manquant
mean(var_quanti) # NA
sd(var_quanti) # NA
var(var_quanti) # NA

## Pour calculer les paramètres après avoir exclu les données manquantes,
## il faut ajouter l'argument 'na.rm = TRUE'
mean(var_quanti, na.rm = TRUE) # 4.7
sd(var_quanti, na.rm = TRUE) # 2.995552
var(var_quanti, na.rm = TRUE) # 8.973333

min(var_quanti, na.rm = TRUE) # 1.3
max(var_quanti, na.rm = TRUE) # 10.4
median(var_quanti, na.rm = TRUE) # 4.25
quantile(var_quanti, probs = c(0, 0.25, 0.5, 0.75, 1), na.rm = TRUE)
```

```
#   0%   25%   50%   75%  100%
# 1.30  2.30  4.25  5.65 10.40

quantile(var_quanti, probs = c(0, 0.25, 0.5, 0.75, 1), na.rm = TRUE, type = 6)
#    0%    25%    50%    75%   100%
# 1.300  1.925  4.250  6.450 10.400
# une autre méthode de calcul des quantiles peut donner des résultats
# légèrement différents (ici type 6 correspond à la méthode de Stata ou de SPSS)

sum(var_quanti, na.rm = TRUE) # 47
prod(var_quanti, na.rm = TRUE) # 722162.4

## La fonction var() applique la formule d'une variance estimée
## à partir d'un échantillon de taille n:
## (somme de (x - moyenne)^2) / (n - 1)
sum((var_quanti[!is.na(var_quanti)] - mean(var_quanti, na.rm = TRUE))^2) / 9
# [1] 8.973333 (ici n-1 = 9, car il y a 10 valeurs non-manquantes)
```

2.5.11.2 Variables qualitatives

Pour décrire une variable qualitative, on peut utiliser les fonctions :

- `table()` qui calcule les effectifs au sein de chaque modalité de réponse
- `prop.table()` ou `proportions()` qui calcule les pourcentages à partir de la fonction `table` précédente.

Les données manquantes sont gérées par l'argument `useNA` avec 3 possibilités :

- `useNA = "no"`, les données manquantes ne sont pas prises en compte (appliqué par défaut)
- `useNA = "ifany"`, les données manquantes sont dénombrées s'il en existe
- `useNA = "always"`, les données manquantes sont toujours dénombrées

```
## on crée 3 variables qualitatives selon 3 formats différents
## (selon un codage numérique, en format caractère, ou en type "factor")
var_quali1 <- c(0, 2, NA, 1, 2, 0, 1, NA, 1, 2, 0, 1, 1)
var_quali2 <- c("homme", "homme", NA, "femme", "homme", "femme", "femme", NA,
                "homme", "femme", "femme", "homme", "femme")
var_quali3 <- factor(c("collège", "univ", "bac", "bac", "univ", NA, "collège",
                       "bac", "bac", "collège", NA, "univ"),
                     levels = c("primaire", "collège", "bac", "univ"))

## la fonction table permet de dénombrer le nombre de réponses par modalité
table(var_quali1)
# var_quali1
# 0 1 2
# 3 5 3
# la valeur 0 apparaît 3 fois, la valeur 1 apparaît 5 fois, la valeur 2 apparaît
# 3 fois

table(var_quali2)
# var_quali2
# femme homme
#     6     5

table(var_quali3)
```

```
# var_quali3
# primaire  collège      bac     univ
#        0        3        4        3
## On note que pour la variable de type "factor", R décrit les résultats pour
## l'ensemble des modalités possibles (ici, 4 modalités possibles, même si
## aucun résultat "primaire" n'a été observé). C'est un des intérêts des
## vecteurs de type "factor".

## Ce ne serait pas le cas avec un vecteur de type caractère :
table(as.character(var_quali3))
# bac collège    univ
#   4       3       3   ici la catégorie "primaire" n'est pas décomptée !
#                       de plus les valeurs sont rangées par ordre alphabétique
#                       plutôt que l'ordre indiqué dans l'attribut 'levels'
#                       du vecteur de type 'factor'.

## Les pourcentages peuvent ensuite être obtenus à partir de ces tables
## avec la fonction prop.table() ou proportions()
prop.table(table(var_quali1))
# var_quali1
#         0         1         2
# 0.2727273 0.4545455 0.2727273   (c'est à dire 27.3%, 45.4% et 27.3%)

prop.table(table(var_quali2))
# var_quali2
#     femme     homme
# 0.5454545 0.4545455       (54.5% de femmes et 45.5% d'hommes)

prop.table(table(var_quali3))
# var_quali3
# primaire  collège      bac     univ
#      0.0      0.3      0.4      0.3     (0%, 30%, 40% et 30%)


## si on applique l'argument useNA = "ifany", les manquants sont considérées
## comme une modalité de réponse en tant que telle
table(var_quali2, useNA = "ifany")
# var_quali2
# femme homme  <NA>    # ici la modalité <NA> est ajoutée
#     6     5     2
proportions(table(var_quali2, useNA = "ifany"))
# var_quali2
#     femme     homme      <NA>
# 0.4615385 0.3846154 0.1538462 # les % sont calculés sur les 3 catégories

## Si on applique l'argument useNA = "always", la catégorie manquante est ajoutée
## comme précédemment
table(var_quali2, useNA = "always")
# var_quali2
# femme homme  <NA>
#     6     5     2

## Si on applique l'argument useNA = "always" a une variable qui n'a pas de
## données manquante, la colonne NA sera quand même ajouté (et compte 0 manquant)
table(c("homme", "homme", "femme", "homme", "femme", "femme",
        "homme", "femme", "femme", "homme", "femme"),
```

```
      useNA = "always")
# femme homme  <NA>
#     6     5     0
```

Il est également utile de connaître la fonction `unique` qui liste les valeurs qui apparaîssent au moins une fois dans un vecteur.

```
## Par exemple avec la variable :
var_quali <- c("homme", "home", NA, "femme", "homme", "femme", "femme", NA,
               "homme", "femme", "Femme", "homme", "femme")
unique(var_quali)
# [1] "homme" "home"  NA      "femme" "Femme"
# on voit que Femme et home font partie des réponses données,
# cela peut correspondre à des erreurs de saisie (une majuscule a été donnée
# à une réponse "femme" et une réponse "homme" a été saisie avec un "m" manquant)
```

2.6 Les listes `list()`

Les **listes** sont des **vecteurs dont les éléments peuvent être de différents types et de différentes dimensions**. Par exemple, une même liste peut contenir des vecteurs, des matrices, des bases de données, des fonctions … (une liste peut même contenir d'autres listes !)

2.6.1 Création d'une liste

Pour créer une liste, on utilise la fonction `list()`. Chaque élément de la liste est séparé par une virgule. Par exemple, on crée ci-dessous un liste qui contient 6 éléments :

1) un vecteur d'entiers, de longueur 10
2) un vecteurs de réels, de longueur 4
3) un vecteurs de chaînes de caractères, de longueur 3
4) un vecteur logique, de longueur 7
5) une matrice de 5 lignes et 2 colonnes
6) une base de données de 3 variables avec 10 valeurs chacune

```
exemple_list <- list(1:10, # un vecteur séquentiel d'entiers, de longueur 10
                     c(1.5, pi, 14, 6.48), # un vecteur de réels
                     c("aa", "hello", "TOULOUSE"), # un vecteur de caractères
                     c(TRUE, TRUE, FALSE, TRUE, FALSE, FALSE, FALSE), # logique
                     matrix(data = 1:10, nrow = 5, ncol = 2), # un matrice
                     data.frame(id = 1:10,                # une base de données
                                sex = c(1,2,1,1,1,2,2,1,2,2),
                                age = c(49,23,43,50,40,37,20,47,26,44)))
exemple_list
# [[1]]
#  [1]  1  2  3  4  5  6  7  8  9 10
#
# [[2]]
# [1]  1.500000  3.141593 14.000000  6.480000
#
# [[3]]
# [1] "aa"       "hello"    "TOULOUSE"
#
# [[4]]
```

```
# [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE
#
# [[5]]
#      [,1] [,2]
# [1,]    1    6
# [2,]    2    7
# [3,]    3    8
# [4,]    4    9
# [5,]    5   10
#
# [[6]]
#    id sex age
# 1   1   1  49
# 2   2   2  23
# 3   3   1  43
# 4   4   1  50
# 5   5   1  40
# 6   6   2  37
# 7   7   2  20
# 8   8   1  47
# 9   9   2  26
# 10 10   2  44
```

2.6.2 Attributs d'une liste

Comme pour les vecteurs atomiques, les listes ont un attribut de classe (`class()`) et de longueur (`length()`).

```
class(exemple_list)
# [1] "list"
length(exemple_list)
# [1] 6               # notre liste d'exemple contient 6 éléments
```

L'attribut `name` est permet de nommer les différents éléments d'une liste.

```
names(exemple_list) <- c("vect_int", "vect_dbl", "vect_char", "vect_logic",
                         "mat", "df")

attributes(exemple_list)
# $names
# [1] "vect_int"   "vect_dbl"   "vect_char"  "vect_logic" "mat"   "df"

exemple_list
# $vect_int                                          # les noms remplacent
#  [1]  1  2  3  4  5  6  7  8  9 10                 # la numérotation
#                                                    # précédente :
# $vect_dbl                                          #  [[1]], [[2]], [[3]],
# [1]  1.500000  3.141593 14.000000  6.480000        #  [[4]], [[5]], [[6]]
#
# $vect_char
# [1] "aa"       "hello"    "TOULOUSE"
#
# $vect_logic
# [1]  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE
#
```

```
# $mat
#      [,1] [,2]
# [1,]    1    6
# [2,]    2    7
# [3,]    3    8
# [4,]    4    9
# [5,]    5   10
#
# $df
#    id sex age
# 1   1   1  49
# 2   2   2  23
# 3   3   1  43
# 4   4   1  50
# 5   5   1  40
# 6   6   2  37
# 7   7   2  20
# 8   8   1  47
# 9   9   2  26
# 10 10   2  44
```

2.6.3 Indexation des éléments d'une liste

On peut sélectionner un seul élément d'une liste :

- par indexation sur la position, avec un double crochet `[[ ]]`
- ou par indexation sur le nom, avec l'opérateur dollar `$`

On peut sélectionner un ou plusieurs éléments d'une liste :

- par indexation sur la position des éléments, avec un simple crochet `[ ]`
- par indexation à l'aide d'un vecteur de noms, avec un simple crochet `[ ]`

```
## Sélection d'un seul élément dans une liste
##  - par indexation sur la position avec le double crochet [[ ]]
exemple_list[[5]]
#      [,1] [,2]
# [1,]    1    6
# [2,]    2    7
# [3,]    3    8
# [4,]    4    9
# [5,]    5   10

##  - par indexation sur le nom, avec l'opérateur dollar $
exemple_list$vect_dbl
# [1]  1.500000  3.141593 14.000000  6.480000

## l'opérateur dollar est une notation abrégée d'une indexation par le nom
##  entre double crochet :
exemple_list[["vect_dbl"]]

## Sélection d'un ou plusieurs éléments dans une liste avec le simple crochet :
## Par exemple, pour sélectionner les élément 1 (vect_int), 2 (vect_dbl),
## et 5 (matrice nommée "mat")
```

```r
exemple_list[c(1, 2, 5)]
# $vect_int
#  [1]  1  2  3  4  5  6  7  8  9 10
#
# $vect_dbl
# [1]  1.500000  3.141593 14.000000  6.480000
#
# $mat
#      [,1] [,2]
# [1,]    1    6
# [2,]    2    7
# [3,]    3    8
# [4,]    4    9
# [5,]    5   10

## On obtient le même résultat avec les noms des éléments, dans un vecteur
## de caractères entre simples crochets :
exemple_list[c("vect_int", "vect_dbl", "mat")]
```

Remarque : les attributs d'un objet sont stockés sous un format de liste

```r
fact_1bis <- factor(c("univ", "bac", "bac", "lycée", "univ", "lycée", "bac"),
                    levels = c("lycée", "bac", "univ"))

attributes(fact_1bis)
# $levels
# [1] "lycée" "bac"   "univ"
#
# $class
# [1] "factor"

length(attributes(fact_1bis))
# [1] 2

class(attributes(fact_1bis))
# [1] "list"

## les attributs de fact_1bis sont une liste contenant 2 vecteurs :
##  - 1 vecteur nommé "levels", de caractères, de longueur 3
##  - 1 vecter nommé "class" contenant la valeur "factor"
```

2.7 Les matrices `matrix()`

Une matrice est une table contenant des données, dont les dimensions sont données par :

- le nombre de lignes (en premier)
- le nombre de colonnes (en deuxième)

Par exemple, une matrice de dimensions (5, 3) est une table de 5 lignes et 3 colonnes.

Dans R, **une matrice est un vecteur avec un attribut de dimensions**, indiquant le nombre de lignes et de colonnes.

2.7.1 Création d'une matrice

On peut créer une matrice à partir d'un vecteur (d'entiers, de réels, de caractères ou de valeurs logiques) auquel on ajoute un attribut de dimension avec la fonction `dim()`.

```
## A partir du vecteur d'entiers de 1 à 15,
seq_1a15 <- 1:15
seq_1a15
# [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
class(seq_1a15)
# [1] "integer"

## on va créer une matrice de dimensions (5, 3) en ajoutant un attribut de
## dimensions avec la fonction dim()
dim(seq_1a15) <- c(5, 3)
seq_1a15
#      [,1] [,2] [,3]
# [1,]    1    6   11
# [2,]    2    7   12
# [3,]    3    8   13
# [4,]    4    9   14
# [5,]    5   10   15
class(seq_1a15)
# [1] "matrix" "array"  # la classe de l'objet est maintenant une matrice

attributes(seq_1a15)    # le vecteur seq_1a15 a un nouvel attribut nommé "dim"
# $dim
# [1] 5 3

## A noter que si l'on attribue une dimension de matrice qui ne correspond pas à
## la longueur du vecteur, cela nous donne un message d'erreur
dim(seq_1a15) <- c(7, 2) # matrice de 7 lignes et 2 colonnes = 14 valeurs
dim(seq_1a15) <- c(4, 4) # matrice de 4 lignes et 4 colonnes = 16 valeurs
```

A noter qu'avec cette méthode, les valeurs ont été rangées par colonne, en 3 colonnes de 5 lignes.

Il est possible d'ajouter un attribut de noms de lignes et un attribut de noms de colonnes à un objet de classe `matrix`, avec les fonctions `rownames()` et `colnames()`.

```
## Ajouter un attribut de nom de lignes :
rownames(seq_1a15) <- c("id_1", "id_2", "id_3", "id_4", "id_5")

## Ajouter un attribut de nom de colonnes :
colnames(seq_1a15) <- c("aa", "bb", "cc")
seq_1a15
#      aa bb cc
# id_1  1  6 11
# id_2  2  7 12
# id_3  3  8 13
# id_4  4  9 14
# id_5  5 10 15

attributes(seq_1a15)
# $dim
# [1] 5 3
#
# $dimnames
```

```
# $dimnames[[1]]
# [1] "id_1" "id_2" "id_3" "id_4" "id_5"
#
# $dimnames[[2]]
# [1] "aa" "bb" "cc"

## Les attributs du vecteur "seq_1a15" sont une liste de deux attributs :
##  1) la dimension de la matrice, nommée "dim"
##  2) les noms des dimensions (nommée "dimnames"), qui est une liste contenant :
##      - le vecteur dimnames[[1]] (noms des lignes)
##      - le vecteur dimnames[[2]] (noms des colonnes)
```

Pour créer une matrice, on peut également utiliser la fonction `matrix()` qui permet d'indiquer directement les arguments :

- `nrow =` et `ncol =` pour définir le nombre de lignes et de colonnes
- `byrow =` qui permet de ranger les valeurs par lignes (en indiquant `TRUE`) ou de les ranger par colonne (en indiquant `FALSE`)
- `dimnames` permet d'indiquer une liste contenant un vecteur de nom de lignes et un vecteur de nom de colonnes.

```
## On utilise la fonction matrix()
mat1 <- matrix(1:15, # le vecteur de valeurs à utiliser
               nrow = 5, # 5 lignes
               ncol = 3, # 3 colonnes
               byrow = FALSE,  # range les valeur par colonne (pas par rang)
               dimnames = list(c("id_1", "id_2", "id_3", "id_4", "id_5"),
                               c("aa", "bb", "cc")))
mat1
#      aa bb cc
# id_1  1  6 11
# id_2  2  7 12
# id_3  3  8 13
# id_4  4  9 14
# id_5  5 10 15

attributes(mat1)
# $dim
# [1] 5 3
#
# $dimnames
# $dimnames[[1]]
# [1] "id_1" "id_2" "id_3" "id_4" "id_5"
#
# $dimnames[[2]]
# [1] "aa" "bb" "cc"

## on peut choisir de ranger les valeurs par rang plutôt que par colonne :
mat2 <- matrix(1:15, # le vecteur de valeurs à utiliser
               nrow = 5, # 5 lignes
               ncol = 3, # 3 colonnes
               byrow = TRUE)  # range les valeur par rang +++
mat2
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    4    5    6
```

```
# [3,]    7    8    9
# [4,]   10   11   12
# [5,]   13   14   15
```

Il est possible de **combiner deux matrices** :

- par rangs avec la fonction `rbind()`
- ou par colonnes avec fonction `cbind()`

```
## Combiner les 2 matrices, mat1 et mat2, par rang :
rbind(mat1, mat2)
#      aa bb cc
# id_1  1  6 11
# id_2  2  7 12
# id_3  3  8 13
# id_4  4  9 14
# id_5  5 10 15
#       1  2  3
#       4  5  6
#       7  8  9
#      10 11 12
#      13 14 15

## Combiner les 2 matrices, mat1 et mat2, par colonne :
cbind(mat1, mat2)
#      aa bb cc
# id_1  1  6 11  1  2  3
# id_2  2  7 12  4  5  6
# id_3  3  8 13  7  8  9
# id_4  4  9 14 10 11 12
# id_5  5 10 15 13 14 15
```

2.7.2 Sélection par indexation au sein d'une matrice

Il est possible de sélectionner un partie de la matrice en indiquant la ligne et la colonne qui nous intéressent, avec la méthode d'indexation par simple crochet `[ , ]`. On indique entre crochets : les lignes que l'on veut sélectionner, suivies d'une virgule, puis les colonnes que l'on veut sélectionner.

L'indexation fonctionne :

- sur la position (en indiquant les numéros de lignes et de colonnes entre crochets)
- ou sur le nom (en indiquant les noms de lignes et de colonnes entre crochets)
- ou sur un vecteur logique pour les lignes et un vecteur logique pour les colonnes

```
## on reprend la matrice mat1 définie précédemment
mat1
#      aa bb cc
# id_1  1  6 11
# id_2  2  7 12
# id_3  3  8 13
# id_4  4  9 14
# id_5  5 10 15

## Pour sélectionner la valeur au croisement de la ligne 3 et de la colonne 2 :
```

```r
mat1[3,2] # [1] 8

## Pour sélectionner une ligne entière, on indique juste une position de ligne
## (valeur avant la virgule entre crochets) :
mat1[3,]
# aa bb cc
#  3  8 13      # c'est la 3ème ligne de la matrice

## Pour sélectionner une colonne entière, on indique juste une position de
## colonne (valeur après la virgule entre crochets) :
mat1[,2]
# id_1 id_2 id_3 id_4 id_5
#    6    7    8    9   10      # valeurs au sein de la 2ème colonne

## Pour sélectionner les valeurs au sein des colonnes 1 et 3,
## et des lignes 3, 4 et 5 :
mat1[3:5, c(1,3)]
#      aa cc
# id_3  3 13
# id_4  4 14
# id_5  5 15

## On peut obtenir les mêmes séléctions en utilisant les noms de lignes et de
## colonnes :
mat1["id_3", "bb"] # croisement de la ligne 3 et de la colonne 2
mat1["id_3",] # sélection de la 3ème ligne
mat1[,"bb"] # sélection de la 2ème colonne
mat1[c("id_3", "id_4", "id_5"), c("aa", "cc")] # lignes 3 à 5 ; colonnes 1 et 3

## On peut obtenir les mêmes séléctions en utilisant des vecteurs logiques :
mat1[c(FALSE, FALSE, TRUE, FALSE, FALSE), c(FALSE, TRUE, FALSE)]
mat1[c(FALSE, FALSE, TRUE, FALSE, FALSE),]
mat1[, c(FALSE, TRUE, FALSE)]
mat1[c(FALSE, FALSE, TRUE, TRUE, TRUE), c(TRUE, FALSE, TRUE)]
```

2.7.3 Collection de plusieurs matrices, `array()` ♠

Il est possible d'ajouter des dimensions à l'attribut `dim()` : cela combine plusieurs matrices de même dimension au sein d'une "collection" de matrices (*array*).

Les dimensions d'un *array* sont définies par la fonction `dim()` :

- les deux premiers chiffres indiquent le nombres de lignes et de colonnes des matrices (comme précédemment)
- le 3ème chiffre indique le nombre de matrices que l'on veut combiner
- il est possible d'ajouter encore d'autres dimensions pour faire des *arrays* dans des *arrays* ...

Par exemple, on peut ranger les valeurs de 1 à 30 dans un *array* de deux matrices de dimensions (5, 3)

```r
## A partir du vecteur d'entiers de 1 à 30,
seq_1a30 <- 1:30

## on va créer un array de dimensions (5, 3, 2) en ajoutant un attribut de
## dimensions avec la fonction dim()
dim(seq_1a30) <- c(5, 3, 2)
seq_1a30
```

```
# , , 1
#
#      [,1] [,2] [,3]
# [1,]    1    6   11
# [2,]    2    7   12
# [3,]    3    8   13
# [4,]    4    9   14
# [5,]    5   10   15
#
# , , 2
#
#      [,1] [,2] [,3]
# [1,]   16   21   26
# [2,]   17   22   27
# [3,]   18   23   28
# [4,]   19   24   29
# [5,]   20   25   30
class(seq_1a30)
# [1] "array"  # la classe de l'objet est maintenant un "array"

## On peut ajouter un attribut de noms aux différentes dimensions à l'aide d'une
## liste de noms :
dimnames(seq_1a30) <- list(noms_rang = c("id_1", "id_2", "id_3", "id_4", "id_5"),
                           noms_col = c("aa", "bb", "cc"),
                           noms_mat = c("mat_1", "mat_2"))

attributes(seq_1a30) # le vecteur seq_1a30 a un attribut de dimension à 3 valeurs
# $dim
# [1] 5 3 2
#
# $dimnames
# $dimnames$noms_rang
# [1] "id_1" "id_2" "id_3" "id_4" "id_5"
#
# $dimnames$noms_col
# [1] "aa" "bb" "cc"
#
# $dimnames$noms_mat
# [1] "mat_1" "mat_2"

seq_1a30 # à présent, les noms de matrices, de rangs et de colonnes apparaîsse
         # à la place des numéros
```

La sélection par indexation fonctionne de la même manière que pour les matrices, mais avec une dimension supplémentaire à ajouter : avec 3 valeurs (ou 3 vecteurs de valeurs) séparées par une virgule entre simples crochets `[ , , ]`

```
## pour sélectionner la valeur à l'intersection de la 4ème ligne et 3ème colonne,
## au sein de la 2ème matrice :
seq_1a30[4,3,2] # [1] 29

## Si on sélectionne uniquement la 4ère ligne des deux matrices
seq_1a30[4,,]
#         noms_mat
# noms_col mat_1 mat_2
#       aa     4    19
```

```
#       bb     9    24
#       cc    14    29
## Le résultat n'est pas intuitifs : les 2 lignes ont été réassemblée sous
## forme de matrice à 2 colonnes !

## On peut sélectionner uniquement la 3ème colonne des deux matrices
seq_1a30[,3,]
#          noms_mat
# noms_rang mat_1 mat_2
#      id_1    11    26
#      id_2    12    27
#      id_3    13    28
#      id_4    14    29
#      id_5    15    30

## On peut sélectionner uniquement la 2ème matrice
seq_1a30[,,2]
#          noms_col
# noms_rang aa bb cc
#      id_1 16 21 26
#      id_2 17 22 27
#      id_3 18 23 28
#      id_4 19 24 29
#      id_5 20 25 30
```

2.8 Les bases de données `data.frame()`

Le format de base des bases de données dans R est le `data.frame()`.

Un `data.frame` correspond à une liste de vecteurs atomiques ayant tous la même longueur (la longueur des vecteurs correspondant au nombre d'individus dans la base de données).

Un *data frame* peut donc être manipulé comme on manipule une liste. Il peut combiner des vecteurs atomiques de différents types (réels, entiers, chaînes de caractères, logiques, facteurs, date, date-heure, etc), et où chaque vecteur peut être nommé (l'attribut `name` permet d'attribuer des noms aux variables).

Vous pouvez récupérer différents exemples de base de données disponibles dans R base, dont la liste est disponible en tapant `library(help = "datasets")`.

```
## Par exemple la base 'women' contient 15 observations de taille (en pouces) et
## poids (en livres) de femmes américaines
?women

women
#    height weight
# 1      58    115
# 2      59    117
# 3      60    120
# 4      61    123
# 5      62    126
# 6      63    129
# 7      64    132
# 8      65    135
# 9      66    139
# 10     67    142
# 11     68    146
```

```
# 12     69    150
# 13     70    154
# 14     71    159
# 15     72    164

class(women)
# [1] "data.frame"
```

2.8.1 Création d'un *data frame*

On peut créer un data.frame contenant différents types de variables avec la fonction `data.frame()`.

```
## Par exemple, on créée une base de données de 10 individus et 4 variables :
df <- data.frame(id = c(1:10),
                 age = c(42.5, 27.9, 60, 74.5, 38, 25.2, 53.4, 46.6, 34.2, 39.6),
                 sex = c("M", "F", "F", "M", "M", "F", "M", "F", "F", "M"),
                 dt_vis = c("2025-12-08", "2025-02-04", "2022-04-22",
                            "2023-10-29", "2023-01-31", "2024-12-17",
                            "2025-09-13", "2025-11-07", "2022-04-25",
                            "2023-08-02"))
df
#    id  age sex     dt_vis
# 1   1 42.5   M 2025-12-08
# 2   2 27.9   F 2025-02-04
# 3   3 60.0   F 2022-04-22
# 4   4 74.5   M 2023-10-29
# 5   5 38.0   M 2023-01-31
# 6   6 25.2   F 2024-12-17
# 7   7 53.4   M 2025-09-13
# 8   8 46.6   F 2025-11-07
# 9   9 34.2   F 2022-04-25
# 10 10 39.6   M 2023-08-02


## Les fonctions head() et tail() permettent de visualiser les premières
## et les dernières lignes d'une base de données
head(df)
#   id  age sex     dt_vis
# 1  1 42.5   M 2025-12-08
# 2  2 27.9   F 2025-02-04
# 3  3 60.0   F 2022-04-22
# 4  4 74.5   M 2023-10-29
# 5  5 38.0   M 2023-01-31
# 6  6 25.2   F 2024-12-17

tail(df)
#    id  age sex     dt_vis
# 5   5 38.0   M 2023-01-31
# 6   6 25.2   F 2024-12-17
# 7   7 53.4   M 2025-09-13
# 8   8 46.6   F 2025-11-07
# 9   9 34.2   F 2022-04-25
# 10 10 39.6   M 2023-08-02

## la fonction View() de R studio permet de visualiser la base de donnée
```

```r
View(df)

## cet objet est de class "data.frame"
class(df)

## Un data.frame est bien un objet de format "list" :
is.list(df) # [1] TRUE

## Un data.frame est une liste qui possède des attributs de noms de variables
## (names) de classe (class) et de noms de rangs (row.names) :
attributes(df)
# $names
# [1] "id"     "age"    "sex"    "dt_vis"
#
# $class
# [1] "data.frame"
#
# $row.names
#  [1]  1  2  3  4  5  6  7  8  9 10

## On peut ainsi récupérer les noms des variables avec la fonction 'names()'
names(df)
# [1] "id"     "age"    "sex"    "dt_vis"
## l'attribut 'names' est un vecteur de caractère dont la longueur est égale
## au nombre de variables

## un data frame a 2 dimensions correspondant au nombre de lignes et de colonnes
## (comme une matrice)
dim(df)
# [1] 10  4          # il y a bien 10 lignes et 4 colonnes
```

2.8.2 Sélection par indexation

On peut sélectionner une variable en appliquant le même principe que pour sélectionner un élément d'une liste :

- par indexation du numéro de la variable entre double crochet `[[ ]]`
- par indexation du nom de la variable entre double crochet `[[ ]]`, ou bien en utilisant l'opérateur dollar `$` suivi du nom de variable

```r
## Sélection par la position de la variable age (2ème variable de la base)
df[[2]]
# [1] 42.5 27.9 60.0 74.5 38.0 25.2 53.4 46.6 34.2 39.6

## Sélection pa le nom de la variable age
df[["age"]]
# [1] 42.5 27.9 60.0 74.5 38.0 25.2 53.4 46.6 34.2 39.6

## Sélection de la variable age par l'opérateur dollar,
df$age # équivalent à df[["age"]]
#[1] 42.5 27.9 60.0 74.5 38.0 25.2 53.4 46.6 34.2 39.6
```

On peut sélectionner le croisement de une (ou plusieurs) ligne(s) et de une (ou plusieurs) colonnes comme pour les matrices, en utilisant les simples crochets où les lignes et les colonnes correspondantes sont séparées par une virgule `[ , ]`

- par indexation sur la position de la ligne et de la colonne,
- par indexation sur le nom de la ligne et le nom de la variable
- ou en utilisant un vecteur logique de conditions

```
## On peut sélectionner la position des lignes 2,4 et 9,
## et des colonnes "age" et "sex" (colonnes 2 et 3)
df[c(2, 4, 9), c(2, 3)]
#    age sex
# 2 27.9   F
# 4 74.5   M
# 9 34.2   F

## On peut également sélectionner sur le noms des lignes et des colonnes
df[c("2", "4", "9"), c("age", "sex")] # donne le même résultat

## On peut mélanger les différents types d'indexation
df[c(2, 4, 9), c("age", "sex")]

## On peut sélectionner les lignes correspondant à une condition,
## par exemple, sélectionner les lignes correspondant aux femmes de plus de 40 ans
df[df$sex == "F" & df$age > 40, c("age", "sex")]
#    age sex
# 3 60.0   F
# 8 46.6   F
```

On peut exclure des lignes ou des colonnes avec un signe “moins” devant les valeurs d’indexation sur la position par ligne ou par colonne :

```
### Exclusion des lignes (2 à 5) et des colonnes (1 et 3)
### avec un signe "moins" devant l'indexation sur la position :
df[-c(2:5), -c(1,3)]
#     age     dt_vis
# 1  42.5 2025-12-08
# 6  25.2 2024-12-17
# 7  53.4 2025-09-13
# 8  46.6 2025-11-07
# 9  34.2 2022-04-25
# 10 39.6 2023-08-02
```

Il est également possible d’utiliser la fonction `subset()`, qui prend pour arguments `subset =` avec des conditions pour sélectionner des lignes (sous la forme d’une condition), et `select =` pour sélectionner des colonnes (avec les noms de colonnes souhaitées).

```
## On peut sélectionner les colonnes id, age et sex,
## chez les femmes de plus de 40 ans
subset(df, # base de donnée
       subset = c(sex == "F" & age > 40), # sous-ensemble de lignes
       select = c(id, age, sex)) # sous-ensemble de colonnes
#   id  age sex
# 3  3 60.0   F
# 8  8 46.6   F
```

2.8.3 Autres formats de bases de données

D’autres formats de bases de données ont été proposés dans R, notamment :

- les `tibble` qui sont associés à l'environement `tidyverse`
- les `data.table` associées au package `data.table`

Pour le début de la formation, nous allons nous concentrer sur les bases au format `data.frame` (qui sont le format R base des bases de données). Nous verrons ensuite comment utiliser les formats `tibble` et `data.table`.

Chapter 3

Analyse simple en R base

Dans ce chapitre, nous allons analyser une base de données simple en utilisant les commandes de R base.

Les objectifs de ce chapitre sont de :

- **préparer un dossier (et un environnement) de travail** pour réaliser les analyses
- **importer des données**, **créer des variables**, **modifier une variable déjà existante**
- réaliser des **analyses descriptives univariées** des variables quantitatives et qualitatives
- réaliser des **analyses descriptives bivariées**
- faire des **représentations graphiques** des données
- savoir faire les **tests statistiques de comparaison** de moyennes et de pourcentages
- savoir faire une **modèle de régression linéaire multivariée**, et vérifier ses conditions d'application
- **sauvegarder** les données et les résultats

Note : Nous verrons dans les chapitre suivants comment réaliser la même analyse en utilisant :

- *des packages spécifiques qui facilitent la réalisation et la présentation des analyses.*
- *la collection de packages du* `tidyverse` *qui apportent de nouvelles fonctions avec une nouvelle philosophie, une nouvelle grammaire et de nouvelles structures de données, qui est très utilisé dans la communauté des utilisateurs de R.*

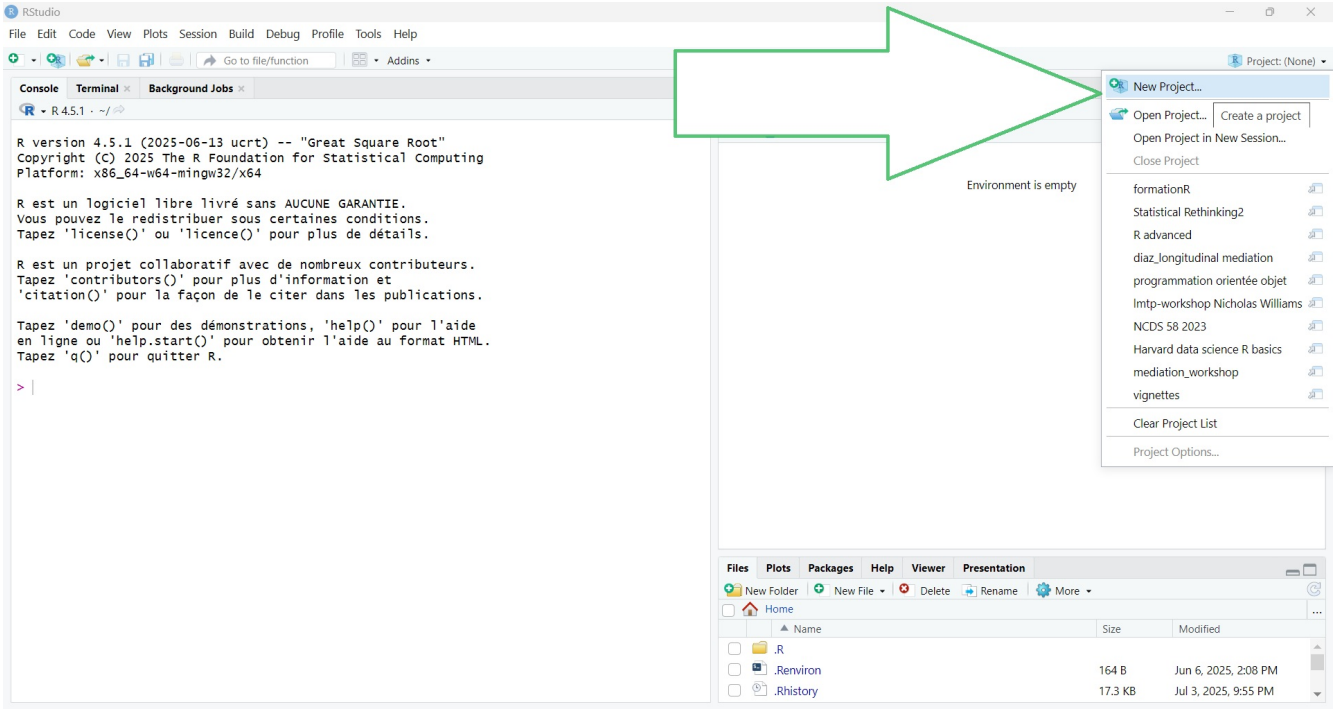
3.1 Préparer un dossier de travail

3.1.1 Projet R

1ère étape : Créer un dossier de travail. Commencez par créer un dossier vide, appelé *analyse R base* dans l'endroit de votre choix sur votre ordinateur. Au sein de ce dossier, vous pouvez ajouter un dossier *data* et un dossier *figures* qui pourra vous servir à stocker les données et les figures que vous souhaitez souvegarder.

2ème étape : Créer un projet R. Dans R Studio, cliquez sur le cube en 3 dimensions R "Project: (None)" en haut à droite, au dessus du cadran environnement.

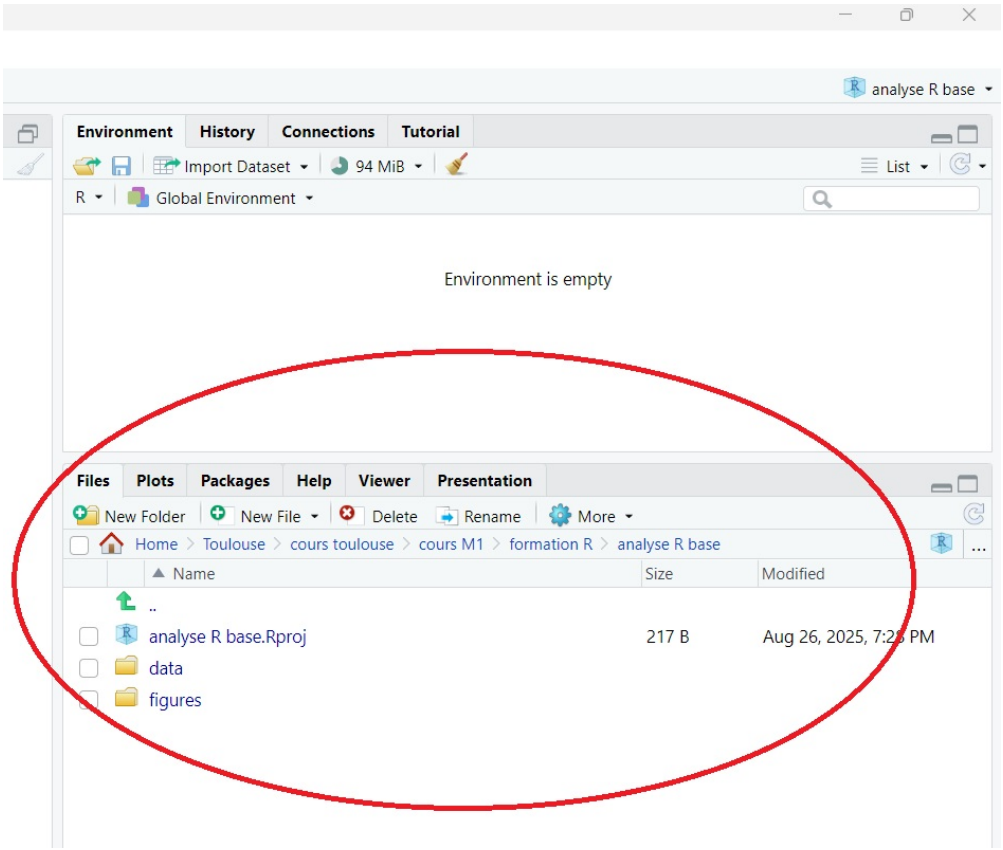
- puis "New Project…".
- choisissez "Existing Directory…". Avec le bouton browse, vous pouvez alors rechercher le dossier *analyse R base* que vous venez de créer à la 1ère étape.
- enfin cliquez sur "Open" et "Create Project".



R Studio va alors créer un fichier `analyse R base.Rproj` au sein de votre dossier *analyse R base*. De plus, ce dossier *analyse R base* sera défini comme votre dossier de travail au sein de R Studio.

Ainsi si vous demandez à R quel est votre dossier de travail actuel avec la commande `getwd()` (*get working directory*), la console va indiquer le chemin jusque ce dossier (R vous confirme qu'il s'agit bien de votre dossier de travail !).

Le contenu de ce dossier de travail apparaît également dans le menu "Files" dans le cadrant en bas à droite. Ce dossier sert de "dossier racine" ("répertoire racine"). Les fichiers au sein de ce dossier pourront être désignés avec un chemin raccourci. **Tous les éléments que vous allez sauvegarder pendant votre session R Studio seront sauvegardés dans ce dossier de travail +++**



Si vous ouvrez un nouveau script R et que vous le sauvegardez (par exemple, sous le nom `1_analyse en R base`), il sera ajouté par défaut au sein de votre dossier de travail, et va apparaître parmi les fichiers du menu file (cadrant en bas à droite).

Maintenant, **quittez le programme R Studio**, puis **double cliquez sur le fichier `analyse R base.Rproj`** (icone d'un cube R en 3 dimensions) qui vient d'être créé dans votre dossier de travail "analyse R base".

R Studio va alors s'ouvrir, en ayant directement défini le dossier *analyse R base* comme dossier de travail. C'est la méthode la plus simple pour reprendre une analyse R en cours et pour reproduire les résultats déjà obtenus.

3.1.2 Structurer un script d'analyse

Lorsque vous réaliserez une analyse, vous allez **sauvegarder vos commandes dans un script** (par exemple, dans le script `1_analyse en R base.R` que l'on vient de sauvegarder précédemment).

En haut du script, si vous souhaitez recommencer à partir d'un environnement vide lorsque vous relancerez l'analyse depuis le départ, vous pouvez ajouter la commande `rm(ls = list())`.

Une bonne pratique est d'indiquer quelques informations utiles pour documenter votre script, comme votre **nom et la date des analyses (en commentaires)**. Enfin, à la fin d'un script (à la fin de vos analyses), vous pouvez lancer la commande `sessionInfo()` qui va lister dans la console la version de R et les versions des packages qui ont été utilisés pour votre analyse (vous pourrez copier ces informations à la fin de votre script, en commentaires).

Les commentaires peuvent également servir à indiquer des **titres** et **sous-titres** dans votre script d'analyse. Un menu d'accès rapide est alors accessible à partir du bouton *Outline* en haut à droite de la fenêtre du script R.

Note : sur windows, le raccourci "ctrl-shift-c" transforme les lignes sélectionnées en commentaires (en ajoutant automatiquement des `#` en début de ligne)

Voici un exemple de script qui suit cette structure :

```r
### 1_analyse en R base
### Date : 15 septembre 2025
### Auteur : Timoté Chalamais

rm(list = ls()) # cette commande vide le contenu de l'environnement

# ---------------------------------------------------------------------------- #
# 1) Titre n°1 (il faut ajouter 4 tirets à la fin des titres) ----
## 1.2) Sous-titre (en ajoutant un # au début de ligne) ----
### 1.2.1) Sous-sous-titre (en ajoutant des # supplémentaires) ----
# ---------------------------------------------------------------------------- #

# Import des données
df_1 <- read.csv2("data/df_1.csv")
meta_df_1 <- read.csv2("data/meta_df_1.csv")

# ---------------------------------------------------------------------------- #
# 2) Titre n°2  ----
# ---------------------------------------------------------------------------- #
head(df_1)

(...) # continuez vous analyses
(...)

# ---------------------------------------------------------------------------- #
# Fin de script  ----

# à la fin, vous pouvez récupérer les informations sur la session
# pour les copier-coller en bas du script (en format de commentaires)
sessionInfo()
# R version 4.5.1 (2025-06-13 ucrt)
# Platform: x86_64-w64-mingw32/x64
# Running under: Windows 11 x64 (build 26100)
#
# Matrix products: default
#   LAPACK version 3.12.1
#
# locale:
# [1] LC_COLLATE=French_France.utf8
# [2] LC_CTYPE=French_France.utf8
# [3] LC_MONETARY=French_France.utf8
# [4] LC_NUMERIC=C
# [5] LC_TIME=French_France.utf8
#
# time zone: Europe/Paris
# tzcode source: internal
#
# attached base packages:
# [1] stats     graphics  grDevices utils     datasets
# [6] methods   base
#
# loaded via a namespace (and not attached):
#  [1] compiler_4.5.1    bookdown_0.43
#  [3] fastmap_1.2.0     cli_3.6.5
#  [5] htmltools_0.5.8.1 tools_4.5.1
#  [7] rstudioapi_0.17.1 yaml_2.3.10
```

```
#  [9] rmarkdown_2.29    knitr_1.50
# [11] xfun_0.52         digest_0.6.37
# [13] rlang_1.1.6       evaluate_1.0.4
```

3.2 Importer une base de données

Pour cet exemple,

- téléchargez la base `df_1.csv` ainsi que la base de méta-données `meta_df_1.csv` (cliquez sur le bouton *download raw file*, à droite avec une flèche vers le bas).
- Vous pouvez ensuite coller ces deux bases de données au sein du dossier *data* de votre dossier de travail.

Il s'agit de fichiers avec l'extension ".csv" (*comma separated variables*) où les valeurs sont séparées par un point virgule. Vous pouvez les importer dans R avec la fonction `read.csv2()` (il existe également une fonction `read.csv()` qui importe les données .csv où le séparateur est une virgule). Vous pouvez regarder les différents arguments de ces fonctions dans l'aide `?read.csv2`.

Nous allons stocker la base dans des objets nommés `df_1` et `meta_df_1`. Ces bases importées sont des objets de type `data.frame`.

```
df_1 <- read.csv2("data/df_1.csv")
meta_df_1 <- read.csv2("data/meta_df_1.csv")
class(df_1)  # [1] "data.frame"
```

- La base `df_1` contient les données que nous allons analyser.
- La base `meta_df_1` contient des méta-données (noms de variables, noms des labels, etc) qui sera utile pour préciser les noms des variables, les labels des variables qualitatives, etc.

3.3 Examiner les données et les méta-données

Comme vu précédemment, vous pouvez examiner les données avec les fonctions `head()`, `tail()`, `str()` et `View()`.

```
head(df_1)
```

```
##   subjid sex  imc trait pas
## 1      1   0 24.8     2 140
## 2      2   0 24.1     3 109
## 3      3   0 26.4     1 156
## 4      4   0 23.3     2 124
## 5      5   0 25.4     2 131
## 6      6   1 25.0     3 148
```

```
tail(df_1)
```

```
##     subjid sex  imc trait pas
## 295    295   1 24.1     1 148
## 296    296   0 18.7     3 121
## 297    297   0 23.3     3 111
## 298    298   1 27.5     3 134
## 299    299   1 24.7     2 158
## 300    300   1 22.8     1 147
```

```r
str(df_1)
```

```
## 'data.frame':    300 obs. of  5 variables:
##  $ subjid: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ sex   : int  0 0 0 0 0 1 0 0 0 0 ...
##  $ imc   : num  24.8 24.1 26.4 23.3 25.4 25 25.2 21.5 21.8 25.9 ...
##  $ trait : int  2 3 1 2 2 3 3 3 1 1 ...
##  $ pas   : int  140 109 156 124 131 148 125 117 132 133 ...
```

```r
View(df_1) # pour voir la base de données dans R Studio
```

On voit que la base de données contient 300 observations et 5 variables :

- `subjid`, l'identifiant patient
- `sex`, le sexe du patient
- `imc`, l'indice de masse corporelle du patient (en kg/m2)
- `trait`, le traitement 0 = placebo, 1 = traitement A, 2 = traitement B
- `pas`, la pression artérielle systolique (en mmHg)

Toutes les variables dans `df_1` sont de type entier (`int`) ou réel (`num`).

```r
meta_df_1 # pour regarder le contenu des méta-données :
```

```
##      var               label id_labs code_labs         labs
## 1 subjid Identifiant patient       1        NA
## 2    sex                Sexe       1         0      Féminin
## 3    sex                Sexe       2         1     Masculin
## 4    imc         IMC (kg/m²)       1        NA
## 5  trait          Traitement       1         1      Placebo
## 6  trait          Traitement       2         2 Traitement A
## 7  trait          Traitement       3         3 Traitement B
## 8    pas          PAS (mmHg)       1        NA
```

La base de méta-données contient 5 variables :

- `var`, nom des variables que l'on retrouve dans la base de données `df_1`
- `label`, nom en détail de la variable

- `id_labs`, un identifiant pour les labels éventuels, par variable
- `code_labs`, codage numérique du label (code utilisé dans la base `df_1`)
- `labs`, modalités de réponses des variables qualitatives, en détail

Vous pouvez également trier les données selon les valeurs d'une ou plusieurs variables avec la fonction `sort_by()` :

```r
### Nous allons trier les données pour nous aider à les inspecter :
### en triant sur le sexe (1ère variable de tri), et l'imc (2ème variable de tri)

### pour voir les première lignes de la base de données triée :
head(sort_by(x = df_1, # x = la base à trier
             y = c(df_1$sex, df_1$imc))) # y = variables de tri, dans l'ordre
```

```
##   subjid sex  imc trait pas
## 1      1   0 24.8     2 140
## 2      2   0 24.1     3 109
## 3      3   0 26.4     1 156
## 4      4   0 23.3     2 124
## 5      5   0 25.4     2 131
## 7      7   0 25.2     3 125
```

```
### pour voir les dernières lignes de la base de donnée triée :
tail(sort_by(x = df_1,
             y = ~ sex + imc), # demandé suivant le format de "formula"
     n = 10) # nombre de lignes à présenter dans les fonctions head() et tail()
```

```
##     subjid sex  imc trait pas
## 258    258   1 29.2     1 142
## 173    173   1 29.5     1 129
## 145    145   1 29.9     2 121
## 92      92   1 30.1     3 167
## 224    224   1 30.2     1 147
## 132    132   1 30.3     2 141
## 144    144   1 30.5     3 158
## 274    274   1 30.5     1 143
## 64      64   1 31.2     1 152
## 223    223   1 31.2     1 172
```

3.4 Créer ou modifier une variable

3.4.1 Créer des variables

Nous allons créer une variable `obesite` dont la valeur est égale à 1 si l'indice de masse corporelle est ≥ 30 kg/m^2 et égale à 0 sinon.

Pour cela, on peut utilise la fonction `ifelse()` dont le premier argument est une condition, le 2ème argument est la valeur à retourner si la condition est vraie, et le 3ème argument est la valeur à retourner si la condition est fausse.

```
df_1$obesite <- ifelse(df_1$imc >= 30, 1, 0)
```

Chaque fois que vous créez ou modifiez une variable, il est utile de vérifier que vous n'avez pas fait d'erreurs. Dans cet exemple, on peut vérifier quelles sont les valeurs minimales et maximales de l'indice de masse corporelle au sein des deux catégories de la nouvelle variable :

```
### vérifier que la variable est correctement crée :
min(df_1$imc[df_1$obesite == 0]) # [1] 15.4
max(df_1$imc[df_1$obesite == 0]) # [1] 29.9
### les valeurs d'IMC varient de 15.4 à 29.9 lorsque obesite == 0

min(df_1$imc[df_1$obesite == 1]) # [1] 30.1
max(df_1$imc[df_1$obesite == 1]) # [1] 32.5
### les valeurs d'IMC varient de 30.1 à 32.5 lorsque obesite == 1 => c'est bon !
```

Nous allons ensuite créer une variable d'IMC en classes (`imc_cl`) définie telle que :

- `imc_cl = 1` (maigreur) si l'IMC < 18.5 kg/m^2,

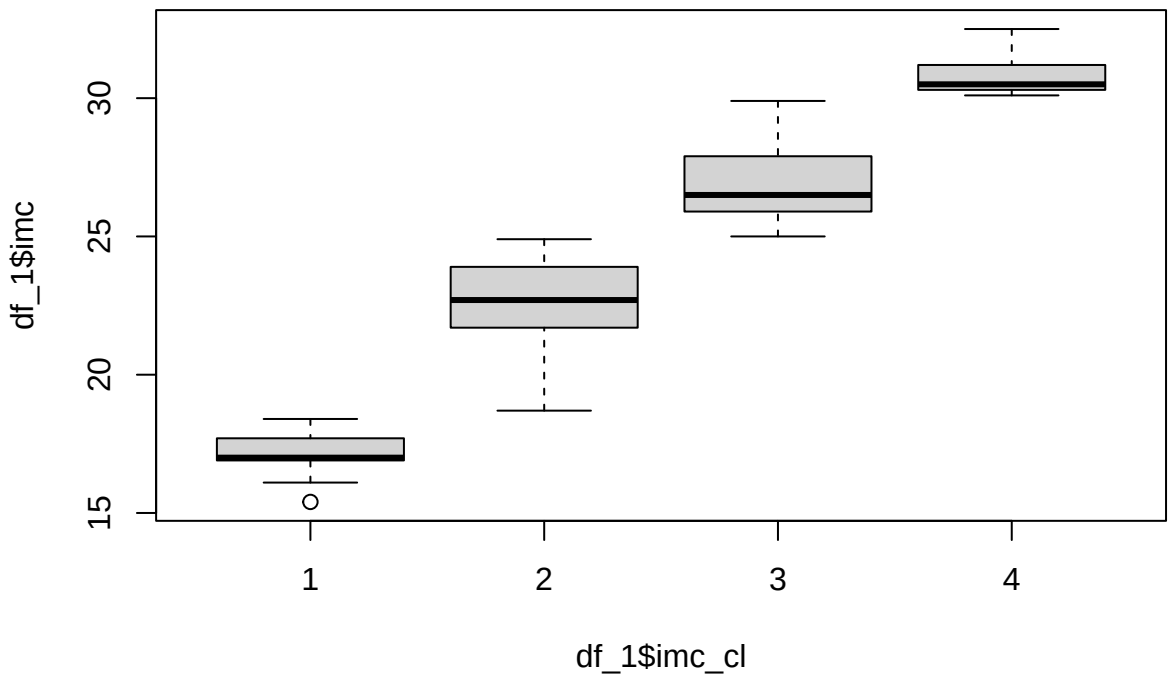
- `imc_cl = 2` (normal) si l'IMC ≥ 18.5 kg/m^2 et IMC < 25 kg/m^2,
- `imc_cl = 3` (normal) si l'IMC ≥ 25 kg/m^2 et IMC < 30 kg/m^2,
- `imc_cl = 4` (normal) si l'IMC ≥ 30 kg/m^2.

Il y a plusieurs possibilités pour créer cette variable, voici 3 méthodes différentes :

```
### 1) créer une variable où toutes les données sont manquantes :
df_1$imc_cl <- rep(NA, nrow(df_1))
###    puis remplacer les données manquantes par les valeurs souhaitées
df_1$imc_cl[df_1$imc < 18.5] <- 1
df_1$imc_cl[df_1$imc >= 18.5 & df_1$imc < 25] <- 2
df_1$imc_cl[df_1$imc >= 25 & df_1$imc < 30] <- 3
df_1$imc_cl[df_1$imc >= 30] <- 4

### 2) avec la fonction ifelse de manière itérative :
df_1$imc_cl <- ifelse(df_1$imc < 18.5, 1,
                      ifelse(df_1$imc >= 18.5 & df_1$imc < 25,
                             2, ifelse(df_1$imc >= 25 & df_1$imc < 30, 3, 4)))

### 3) avec une formule intégrant des conditions.
###    Comme on multiplie les résultats de conditions à des entiers, les
###    réponses TRUE et FALSE sont transformées en 1 et 0 par coercition
df_1$imc_cl <- (1 * (df_1$imc < 18.5) +
                  2 * (df_1$imc >= 18.5 & df_1$imc < 25) +
                  3 * (df_1$imc >= 25 & df_1$imc < 30) +
                  4 * (df_1$imc >= 30))
```

Pensez à vérifier que vous n'avez pas fait d'erreur en créant cette variable. Par exemple, on peut vérifier la distribution de l'IMC au sein des 4 classes à l'aide d'un box plot.

```
boxplot(df_1$imc ~ df_1$imc_cl) # imc en fonction de l'imc en classe
```



Après avoir créé ces deux nouvelles variables, nous pouvons compléter la base de méta-données pour définir les noms de variables et de labels en clair. Pour cela, nous allons ajouter des lignes supplémentaires avec la fonction `rbind()` (*row bind* pour "fusion par rang"). *A noter qu'il existe aussi une fonction* `cbind()` *qui permet de fusionner des bases ou des matrices par colonnes*

```
meta_df_1 <- rbind(meta_df_1,
                   data.frame(var = c(rep("obesite", 2),
                                      rep("imc_cl", 4)),
                              label = c(rep("Obésité", 2),
                                        rep("IMC en classes", 4)),
                              id_labs = c(1:2, 1:4),
                              code_labs = c(0:1, 1:4),
                              labs = c("Non", "Oui",
                                       "Maigreur", "Normal", "Surpoids", "Obèse"))
                   )
meta_df_1
```

```
##        var               label id_labs code_labs         labs
## 1   subjid Identifiant patient       1        NA
## 2      sex                Sexe       1         0      Féminin
## 3      sex                Sexe       2         1     Masculin
## 4      imc         IMC (kg/m²)       1        NA
## 5    trait          Traitement       1         1      Placebo
## 6    trait          Traitement       2         2 Traitement A
## 7    trait          Traitement       3         3 Traitement B
## 8      pas          PAS (mmHg)       1        NA
## 9  obesite             Obésité       1         0          Non
## 10 obesite             Obésité       2         1          Oui
```

```
## 11  imc_cl      IMC en classes       1         1     Maigreur
## 12  imc_cl      IMC en classes       2         2       Normal
## 13  imc_cl      IMC en classes       3         3     Surpoids
## 14  imc_cl      IMC en classes       4         4        Obèse
```

3.4.2 Modifier des variables

Enfin, nous souhaitons modifier la valeur de la pression artérielle du patient n°137 : suite à une erreur de saisie, nous nous sommes rendu compte que sa valeur n'est pas 133 mmHg, mais 123 mmHg.

```
df_1$pas[df_1$subjid == 137]
```

```
## [1] 133
```

```
## nous pouvons directement assigner la nouvelle valeur avec l'indexation par
## condition :
df_1$pas[df_1$subjid == 137] <- 123
df_1[df_1$subjid %in% 135:140,]
```

```
##     subjid sex  imc trait pas obesite imc_cl
## 135    135   0 28.0     2 154       0      3
## 136    136   1 26.2     1 138       0      3
## 137    137   0 20.1     1 123       0      2
## 138    138   0 24.7     3 155       0      2
## 139    139   0 17.0     2 115       0      1
## 140    140   1 25.7     1 160       0      3
```

```
# on voit que la valeur du patient 137 a bien été corrigée
```

3.4.3 Sauvegarder la base de données

Pour sauvegarder la base de données avec les variables que vous venez de créer et de modifier, vous pouvez utiliser les fonctions :

- `write.csv2()` sauvegarde une base au format ".csv" avec un point-virgule comme séparateur (comme pour `read`, il existe une fonction `write.csv2()` qui utilise une virgule comme séparateur). Le format .csv peut être lu par tous les logiciels statistiques, les gestionnaires de base de données (commme Access, LibreOffice Base), les tableurs (comme Excel, LibreOffice Calc, …) ou tout éditeur de texte (bloc-note, TextEdit, etc).
- `saveRDS()` permet de sauvegarder tout objet R. Il sera sauvegardé avec l'extension ".R". Vous pourrez seulement l'importer dans le logiciel R avec la commande `readRDS()`. L'intérêt de ce format est que l'objet R est sauvegardé avec tous ses attributs R.

```
### Sauvegarder la base df_1 dans le dossier data, au format .csv,
### pensez à lui donnant un nouveau nom "df_1_new.csv" pour ne pas écraser
### l'ancienne base "df_1.csv"
write.csv2(df_1, "data/df_1_new.csv")

### Sauvegarder la base df_1 dans le dossier data, au format .R,
### pensez à lui donnant un nouveau nom "df_1_new.R" pour ne pas écraser
### l'ancienne base "df_1.csv"
saveRDS(df_1, "data/df_1_new.R")
```

```
### pour importer à nouveau ces objets dans R :
df_1_new <- read.csv2("data/df_1_new.csv")
df_1_new_bis <- readRDS("data/df_1_new.R")
# vous retrouvez alors la base dans l'environnement avec leur nouveau nom :
# df_1_new et df_1_new_bis
```

3.5 Analyses univariées

3.5.1 fonction `summary()`

Nous pouvons utiliser la fonction `summary()` pour obtenir de manière synthétique une description univariée de l'ensemble des variables, indiquant : les valeurs minimales et maximales, quartiles et médiane, et la valeur moyenne.

```
summary(df_1)
```

```
##      subjid            sex            imc            trait
##  Min.   :  1.00   Min.   :0.00   Min.   :15.40   Min.   :1.000
##  1st Qu.: 75.75   1st Qu.:0.00   1st Qu.:22.30   1st Qu.:1.000
##  Median :150.50   Median :0.00   Median :24.60   Median :2.000
##  Mean   :150.50   Mean   :0.49   Mean   :24.48   Mean   :1.897
##  3rd Qu.:225.25   3rd Qu.:1.00   3rd Qu.:26.40   3rd Qu.:3.000
##  Max.   :300.00   Max.   :1.00   Max.   :32.50   Max.   :3.000
##       pas           obesite           imc_cl
##  Min.   : 92.0   Min.   :0.00000   Min.   :1.000
##  1st Qu.:125.0   1st Qu.:0.00000   1st Qu.:2.000
##  Median :138.0   Median :0.00000   Median :2.000
##  Mean   :137.1   Mean   :0.04333   Mean   :2.453
##  3rd Qu.:149.0   3rd Qu.:0.00000   3rd Qu.:3.000
##  Max.   :177.0   Max.   :1.00000   Max.   :4.000
```

Les variables de la base `df_1` sont toutes codées en valeurs numériques, la fonction summary a donc décrit les variables qualitatives comme s'il s'agissait de variables quantitatives : ce n'est pas adapté.

Nous allons créer 2 nouvelles variables en `factor` à partir des variables qualitatives. Pour cela, nous allons également nous servir des informations indiquées dans la base de méta-données. Puis nous allons relancer la fonction `summary()` :

```
### création de 2 variables de type "factor" à partir des variables sex et trait
df_1$sexL <- factor(df_1$sex,
                    labels = meta_df_1$labs[meta_df_1$var == "sex"])
df_1$traitL <- factor(df_1$trait,
                      labels = meta_df_1$labs[meta_df_1$var == "trait"])
summary(df_1)
```

```
##      subjid            sex            imc            trait
##  Min.   :  1.00   Min.   :0.00   Min.   :15.40   Min.   :1.000
##  1st Qu.: 75.75   1st Qu.:0.00   1st Qu.:22.30   1st Qu.:1.000
##  Median :150.50   Median :0.00   Median :24.60   Median :2.000
##  Mean   :150.50   Mean   :0.49   Mean   :24.48   Mean   :1.897
##  3rd Qu.:225.25   3rd Qu.:1.00   3rd Qu.:26.40   3rd Qu.:3.000
##  Max.   :300.00   Max.   :1.00   Max.   :32.50   Max.   :3.000
##       pas           obesite           imc_cl           sexL
##  Min.   : 92.0   Min.   :0.00000   Min.   :1.000   Féminin :153
```

```
##  1st Qu.:125.0   1st Qu.:0.00000   1st Qu.:2.000   Masculin:147
##  Median :138.0   Median :0.00000   Median :2.000
##  Mean   :137.1   Mean   :0.04333   Mean   :2.453
##  3rd Qu.:149.0   3rd Qu.:0.00000   3rd Qu.:3.000
##  Max.   :177.0   Max.   :1.00000   Max.   :4.000
##           traitL
##  Placebo     :120
##  Traitement A: 91
##  Traitement B: 89
##
##
##
```

Les nouvelles variables `sexL` et `traitL` sont à présent décrites par dénombrement du nombre d'individus par modalité de réponse, de manière adaptée aux variables qualitatives.

3.5.2 Variables quantitatives

Pour décrire des variables quantitatives, on peut s'intéresser aux paramètres suivants :

- effectifs observés (non-manquants), avec les fonctions `length()` qui indique la longueur d'un vecteur, ou `nrow()` qui indique le nombre de lignes d'une base de données (ou d'une matrice)
- moyenne, avec la fonction `mean()`
- écart type et variance, avec les fonctions `sd()` et `var()`
- minimum, 1er quartiles, médiane, 3ème quartile, maximum, avec les fonctions `min()`, `max()`, `median()`, `quantiles()`

```r
### Effectifs observés :
### Comme il n'y a pas de manquant dans cette base, on peut directement utiliser
### la fonction length() ou nrow() pour connaître les effectifs
nrow(df_1) # 300
length(df_1$imc) # 300

### Pour compter les effectifs non-manquants, de manière explicite :
length(df_1$imc[!is.na(df_1$imc)]) # 300

### moyennes
mean(df_1$imc, na.rm = TRUE) # 24.481
mean(df_1$pas, na.rm = TRUE) # 137.1133

### déviation standard (écart-type)
sd(df_1$imc, na.rm = TRUE) # 3.069072
sd(df_1$pas, na.rm = TRUE) # 16.82053

### variances
var(df_1$imc, na.rm = TRUE) # 9.419203
var(df_1$pas, na.rm = TRUE) # 282.9303

### quantiles
min(df_1$imc, na.rm = TRUE) # 15.4
min(df_1$pas, na.rm = TRUE) # 92

max(df_1$imc, na.rm = TRUE) # 32.5
max(df_1$pas, na.rm = TRUE) # 177
```

```r
median(df_1$imc, na.rm = TRUE) # 24.6
median(df_1$pas, na.rm = TRUE) # 138

quantile(df_1$imc, probs = c(0, 0.25, 0.5, 0.75, 1), na.rm = TRUE)
#   0%  25%  50%  75% 100%
# 15.4 22.3 24.6 26.4 32.5
quantile(df_1$pas, probs = c(0, 0.25, 0.5, 0.75, 1), na.rm = TRUE)
  # 0%  25%  50%  75% 100%
  # 92  125  138  149  177
```

3.5.3 Programmation élémentaire

3.5.3.1 Créer une nouvelle fonction

Il est fastidieux de devoir récupérer les différents paramètres de distribution de chaque variable, un par un !

Mais R permet facilement de programmer de nouvelles fonctions "maison" à l'aide de `function(arguments) { expression }`, où on indique une liste d'`arguments` entre parenthèses, puis une liste de commandes à réaliser entre les accolades.

Par exemple, on peut créer une nouvelle fonction `exemple_fonction()` qui ajoute +2 à l'objet `x` :

```r
seq_1a5 <- c(1:5)
exemple_fonction <- function(x) {return(x + 2)}
exemple_fonction(seq_1a5)
```

```
## [1] 3 4 5 6 7
```

Pour accélérer notre analyse des variables quantitatives, nous allons créer **une nouvelle fonction qui va calculer d'un coup l'ensemble des paramètres qui nous intéressent**. Cette fonction sera nommée `univ_quanti`, elle va dépendre de 3 arguments :

- `x`, la variable à décrire
- `dig`, le nombre de chiffres après la virgule pour présenter des valeurs arrondies
- `remove_miss`, une valeur logique qui sera utilisée pour exclure (ou non) les données manquantes du calcul (pour renseigner l'argument `na.rm` des fonctions descriptives de base)

Entre accolades, on va demander à la fonction de :

- calculer les effectifs, la moyenne, l'écart type et les quantiles, de la variable `x`
- regrouper ces valeurs au sein d'un vecteur de réels, arrondis à `dig` chiffres après la virgule,
- retourner le vecteur obtenu

```r
### Définir une nouvelle fonction dans R
univ_quanti <- function(x, # la variable à décrire
                        dig = 2, # par défaut, 2 chiffres après la virgule
                        remove_miss = TRUE # par défaut, la valeur est TRUE
                        ) { # fermez la parenthèse et ouvrez l'accolade
  # on commence par calculer les différents paramètres et on les stocke dans
  # les objets : "n", "moy", "sd" et "q"
  n <- length(x[!is.na(x)])
  moy <- mean(x, na.rm = remove_miss)
  sd <- sd(x, na.rm = remove_miss)
  q <- quantile(x, probs = c(0, 0.25, 0.5, 0.75, 1), na.rm = remove_miss)
```

```
  # on stocke les résultats dans un vecteur de réels, nommé "param",
  # en gardant uniquement la valeur arrondie pour la moyenne et l'écart-type
  param <- c(n,
             round(moy, digits = dig),
             round(sd, digits = dig),
             q)
  # on peut ajouter un nom à chaque élément du vecteur "param"
  names(param) <- c("N", "mean", "sd", "min", "Q1", "median", "Q3", "max")

  # indiquer ce que doit retourner la fonction
  return(param)
} # fermez l'accolade
```

Si on applique cette fonction à nos deux variables quantitatives, on obtient l'ensemble des paramètres présentés dans un vecteur :

```
univ_quanti(df_1$imc, dig = 1, remove_miss = TRUE)
```

```
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0   24.5    3.1   15.4   22.3   24.6   26.4   32.5
```

```
univ_quanti(df_1$pas, dig = 1, remove_miss = TRUE)
```

```
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0  137.1   16.8   92.0  125.0  138.0  149.0  177.0
```

Note : Vous pouvez remarquer qu'en utilisant cette fonction, les objets `n`*,* `moy`*,* `sd`*,* `q` *et* `param` *n'apparaissent pas dans l'environnement de travail, ils ont uniquement été créés de manière temporaire au sein de la fonction.*

3.5.3.2 Utiliser des boucles

Les boucles `for` dans R permettent également d'automatiser des opérations en boucle. La syntaxe d'une boucle est :

- `for (variable in sequence) { expression }` où l'`expression` entre accolade est répétée chaque fois que la `variable` est égale à une valeur de la `sequence`

```
### exemple de boucle :
###  - pour chaque valeur i variant de 1 à 5,
###    => calcule 20 + i
###    => ajoute cette valeur à la fin de la phrase "calcul de 20 + i = "
###    => imprime le résultat à l'écran
for (i in 1:5) {
  print(paste0("calcul de 20 + i = ", 20 + i))
}
```

```
## [1] "calcul de 20 + i = 21"
## [1] "calcul de 20 + i = 22"
## [1] "calcul de 20 + i = 23"
## [1] "calcul de 20 + i = 24"
## [1] "calcul de 20 + i = 25"
```

On peut appliquer cette démarche pour répéter l'analyse univariée avec notre nouvelle fonction `univ_quanti()` à nos deux variables quantitatives. Les deux variables quantitatives sont dans les colonnes 3 et 5 de la base de données.

```
### Avec la fonction names(), on voit que les variables quantitatives imc et pas
### sont la 3ème et la 5ème variable de la base df_1
names(df_1)
```

```
## [1] "subjid"  "sex"     "imc"     "trait"   "pas"     "obesite" "imc_cl"
## [8] "sexL"    "traitL"
```

```
for (i in c(3, 5)) {
  print(names(df_1)[i]) # imprime le nom de la i-ème variable
  # puis imprime les résultats de la fonction univ_quanti appliquée à la i-ème
  # variable de la base df_1
  print(univ_quanti(df_1[[i]], dig = 1, remove_miss = TRUE))
}
```

```
## [1] "imc"
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0   24.5    3.1   15.4   22.3   24.6   26.4   32.5
## [1] "pas"
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0  137.1   16.8   92.0  125.0  138.0  149.0  177.0
```

Question bonus : que se passe t'il si vous n'indiquez pas la fonction `print()` ?

- *réponse : il a fait tourner la fonction, mais ne l'a pas imprimé les résultats à l'écran*

3.5.3.3 Fonctions `apply()`, `lapply()`, `sapply()`

On peut également lancer une fonction de manière répétée appliquée :

- à des colonnes ou des lignes de matrices (ou de data frame) avec la fonction `apply()`
- à des vecteurs ou des listes avec `lapply()` (qui retourne les résultats sous forme de liste) ou `sapply()` (qui retourne les résultats sous forme de matrice ou de vecteur)

Exemple 1 : La fonction `apply()` permet d'appliquer notre fonction de description des paramètres aux colonnes `"imc"` et `"pas"` de la la base `df_1` (Le résultat sera une matrice de réels) :

```
apply(df_1[,c("imc", "pas")], # matrice ou data.frame sélectionnée
      MARGIN = 2, # 2 = par colonne ;  1 = par ligne
      FUN = univ_quanti, # fonction à utiliser
      dig = 1, # on peut ajouter les arguments de la fonction à la suite
      remove_miss = TRUE)
```

```
##          imc   pas
## N      300.0 300.0
## mean    24.5 137.1
## sd       3.1  16.8
## min     15.4  92.0
## Q1      22.3 125.0
## median  24.6 138.0
## Q3      26.4 149.0
## max     32.5 177.0
```

Exemple 2 : La fonction `lapply()` applique la fonction à une liste de vecteurs et retourne une liste de la même longueur. La fonction `sapply()` fait la même chose, mais retourne les résultats sous forme de vecteur ou de matrice.

```
lapply(X = df_1[,c("imc", "pas")],
      FUN = univ_quanti,  # fonction à utiliser
      dig = 1,
      remove_miss = TRUE)
```

```
## $imc
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0   24.5    3.1   15.4   22.3   24.6   26.4   32.5
##
## $pas
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0  137.1   16.8   92.0  125.0  138.0  149.0  177.0
```

```
sapply(X = df_1[,c("imc", "pas")],
       FUN = univ_quanti,  # fonction à utiliser
       dig = 1,
       remove_miss = TRUE)
```

```
##          imc   pas
## N      300.0 300.0
## mean    24.5 137.1
## sd       3.1  16.8
## min     15.4  92.0
## Q1      22.3 125.0
## median  24.6 138.0
## Q3      26.4 149.0
## max     32.5 177.0
```

Si on veut remplacer les noms de colonnes par les noms de variables en clair, on peut utiliser la base de méta-données :

```
res_quanti <- sapply(df_1[,c("imc", "pas")],
                     FUN = univ_quanti,
                     dig = 1)
# on modifie le nom des colonne de la matrice de résultats
# avec les informations disponibles dans les méta-données
colnames(res_quanti) <- c(meta_df_1$label[meta_df_1$var == "imc"],
                          meta_df_1$label[meta_df_1$var == "pas"])
res_quanti
```

```
##        IMC (kg/m²) PAS (mmHg)
## N            300.0      300.0
## mean          24.5      137.1
## sd             3.1       16.8
## min           15.4       92.0
## Q1            22.3      125.0
## median        24.6      138.0
## Q3            26.4      149.0
## max           32.5      177.0
```

3.5.4 Variables qualitatives

Pour décrire les variables qualitatives,

- la fonction `table()` permet de décrire les effectifs dans chaque modalité de réponse,
- la fonction `prop.table()` permet de décrire les pourcentages des données obtenues avec la fonction `table()`,

On va utiliser les variables en "facteur" `sexL` et `traitL` dont les modalités de réponse sont labellisées.

```
### Description de la variable sex
table(df_1$sexL) # retourne un vecteur avec les effectifs
```

```
##
##  Féminin Masculin
##      153      147
```

```
prop.table(table(df_1$sexL)) # retourne un vecteur avec les pourcentages
```

```
##
##  Féminin Masculin
##     0.51     0.49
```

```
### On peut combiner ces deux vecteurs avec cbind() (combinaison par colonne)
### pour les afficher dans une matrice
tab_sex <- cbind(table(df_1$sexL),
                 round(prop.table(table(df_1$sexL)) * 100, digits = 1))
colnames(tab_sex) <- c("n", "pct")
tab_sex
```

```
##            n pct
## Féminin  153  51
## Masculin 147  49
```

```
### De même pour la variable traitement
tab_trait <- cbind(table(df_1$traitL),
                   round(prop.table(table(df_1$traitL)) * 100, digits = 1))
colnames(tab_trait) <- c("n", "pct")
tab_trait
```

```
##                n  pct
## Placebo      120 40.0
## Traitement A  91 30.3
## Traitement B  89 29.7
```

3.6 Représentations graphiques

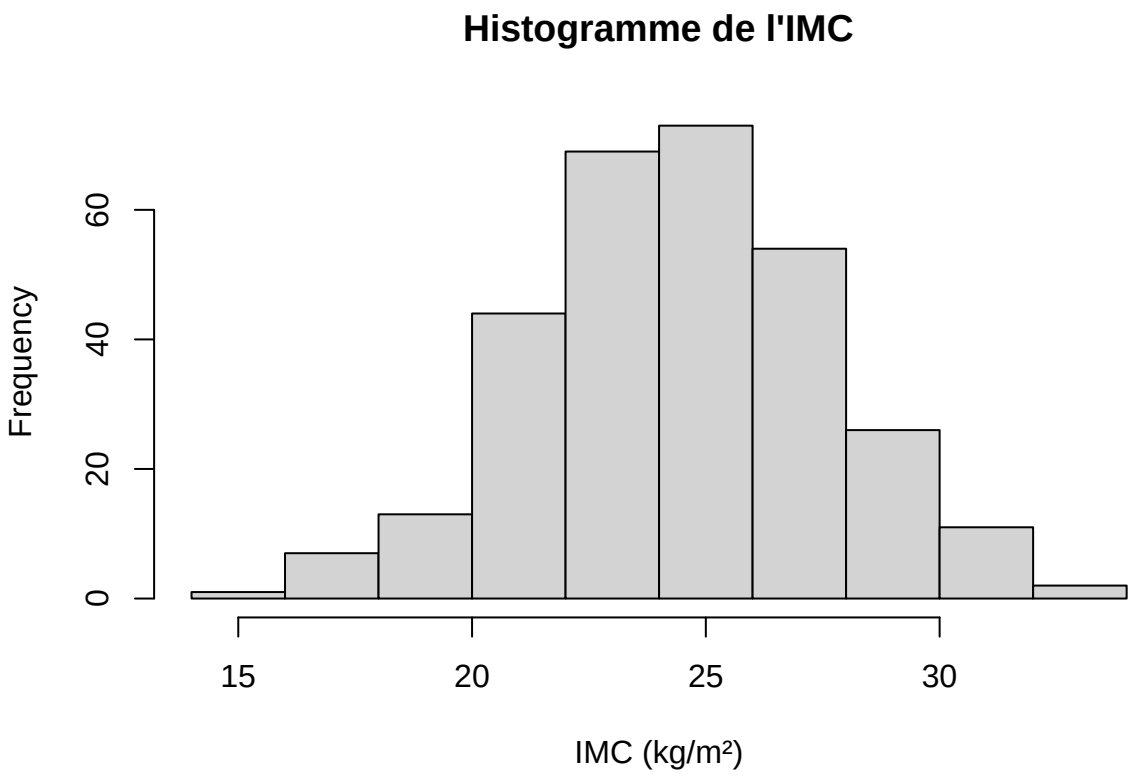
Il est toujours utile de faire des représentations graphiques de la distribution des variables : cela permet d'avoir une bonne vision de l'ensemble des données et de détecter des anomalies éventuelles.

3.6.1 Distributions univariées

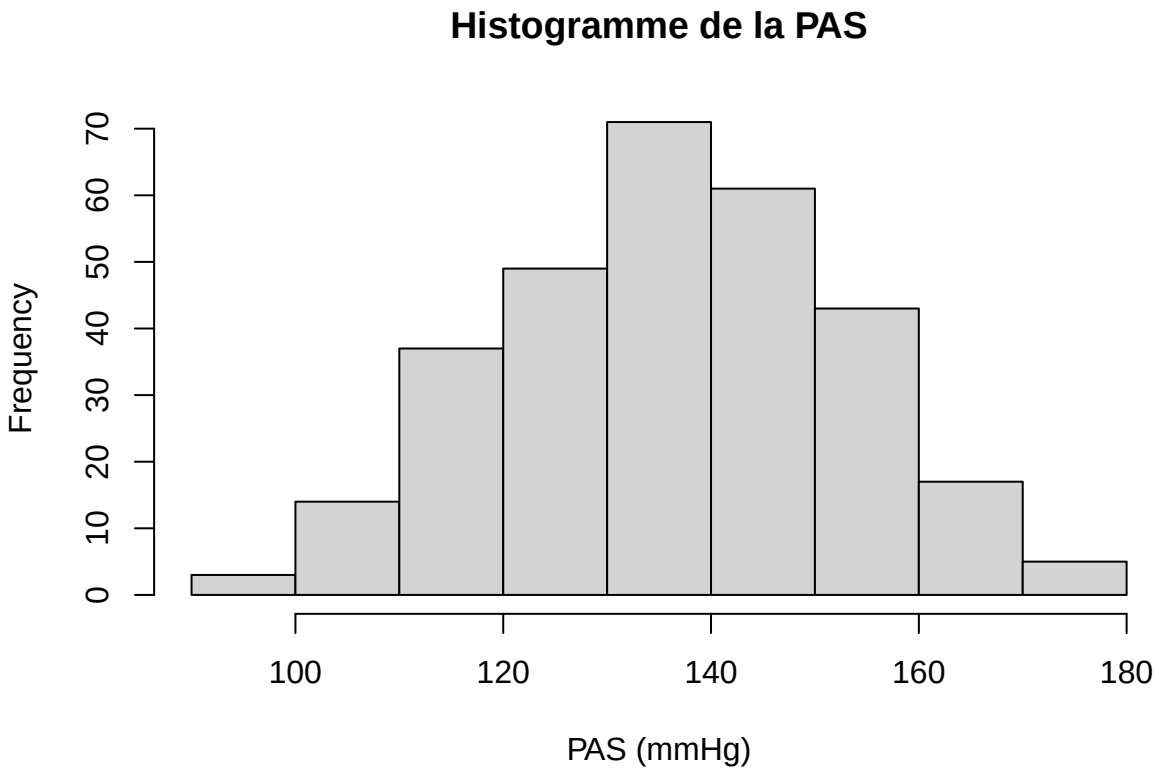
La distribution des variables quantitatives peut être représentée à l'aide d'un histogramme `hist()`, d'une densité de kernel `density()` ou d'un box plot `boxplot()`.

Note : On peut modifier les paramètres des graphiques avec la fonction `par`. Par exemple `mfrow()` permet de combiner des figures par lignes et par colonnes.

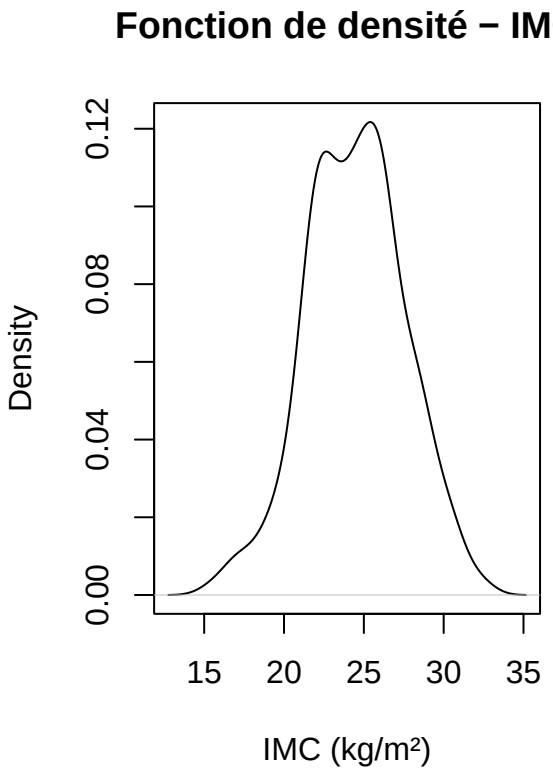
```
### Variables quantitatives : imc et pas
### Histogrammes
hist(df_1$imc, xlab = "IMC (kg/m²)", main = "Histogramme de l'IMC")
```

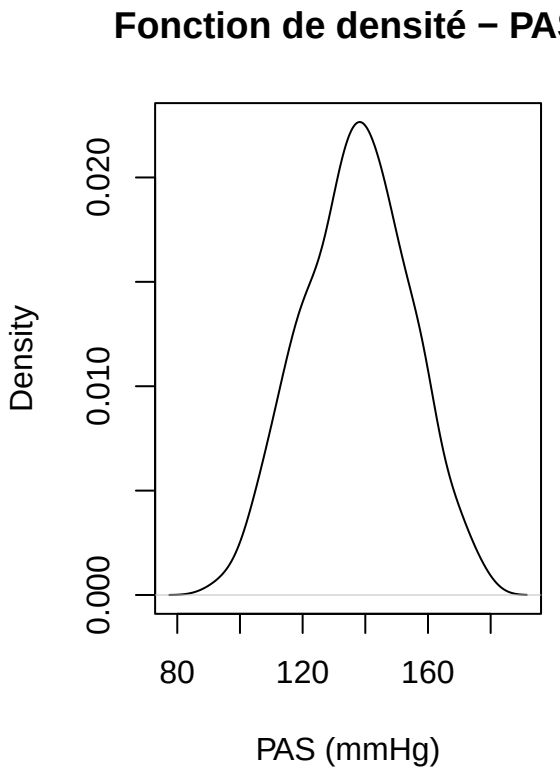



```
hist(df_1$pas, xlab = "PAS (mmHg)", main = "Histogramme de la PAS")
```

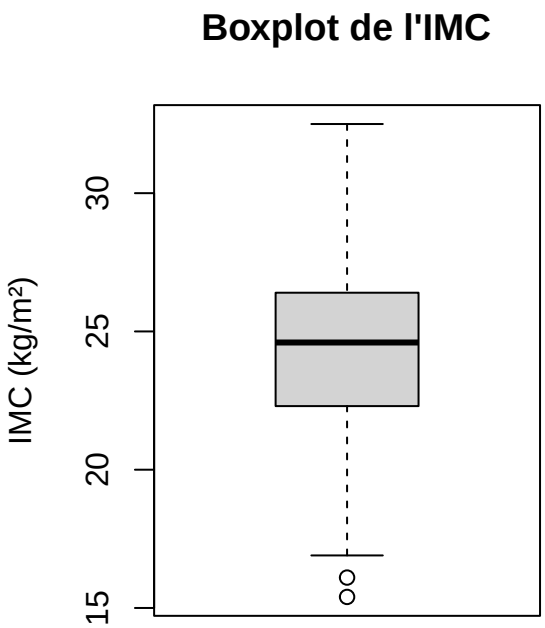


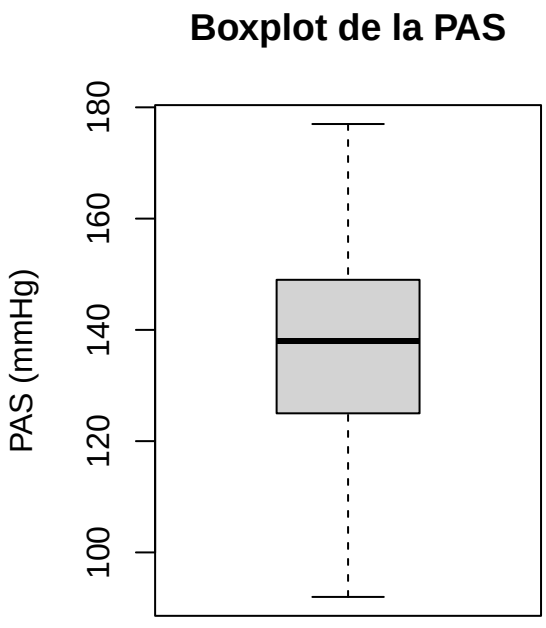

```r
### On va afficher deux figures sur une 1 ligne et deux colonnes
par(mfrow = c(1, 2)) # indiquer c(X, Y) ou X = nb de lignes et Y = nb de colonnes

### Densités de kernel
plot(density(df_1$imc), xlab = "IMC (kg/m²)", main = "Fonction de densité - IMC")
plot(density(df_1$pas), xlab = "PAS (mmHg)", main = "Fonction de densité - PAS")
```





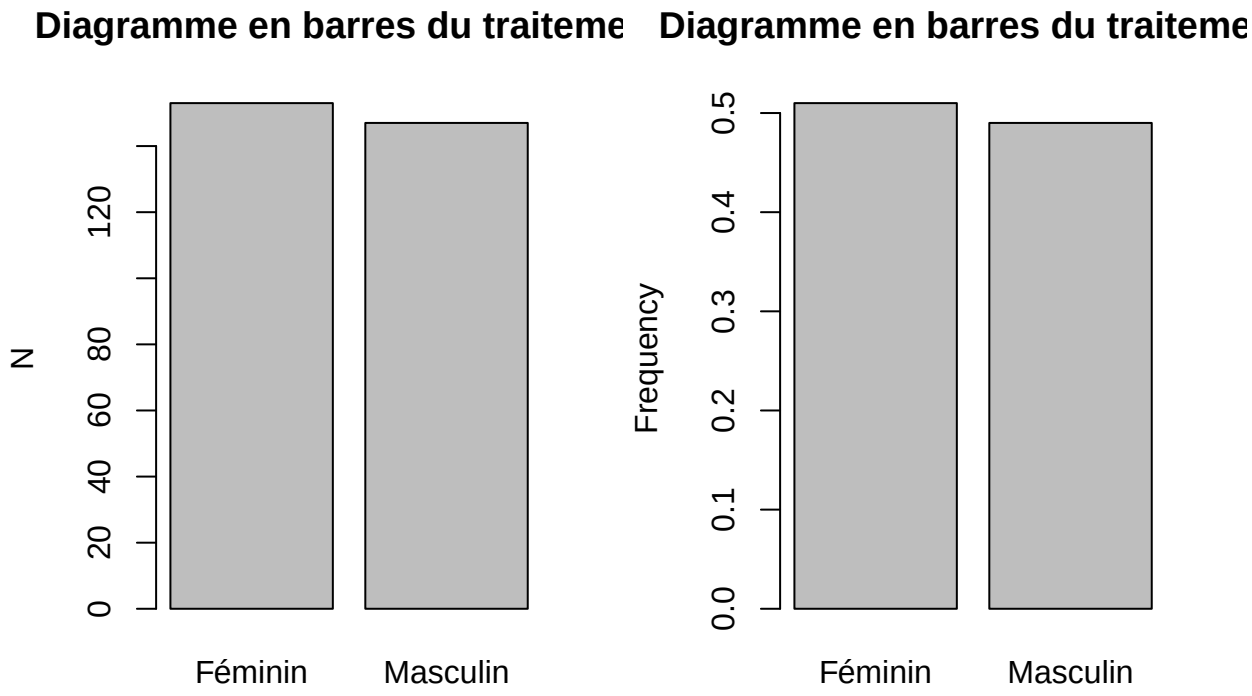

```r
### Box plots
boxplot(df_1$imc, main = "Boxplot de l'IMC", ylab = "IMC (kg/m²)")
boxplot(df_1$pas, main = "Boxplot de la PAS", ylab = "PAS (mmHg)")
```





Les distributions des variables qualitatives peuvent être représentées par des diagrammes en barres `barplot()`.

```r
### Diagrammes en barres, appliqués aux "facteurs"
par(mfrow = c(1, 2)) # 2 figures sur une seule ligne
barplot(table(df_1$sexL), # appliquer la fonction à une table()
        ylab = "N",
        main = "Diagramme en barres du traitement") # avec les effectifs
barplot(prop.table(table(df_1$sexL)),
        ylab = "Frequency",
        main = "Diagramme en barres du traitement") # avec les pourcentages
```

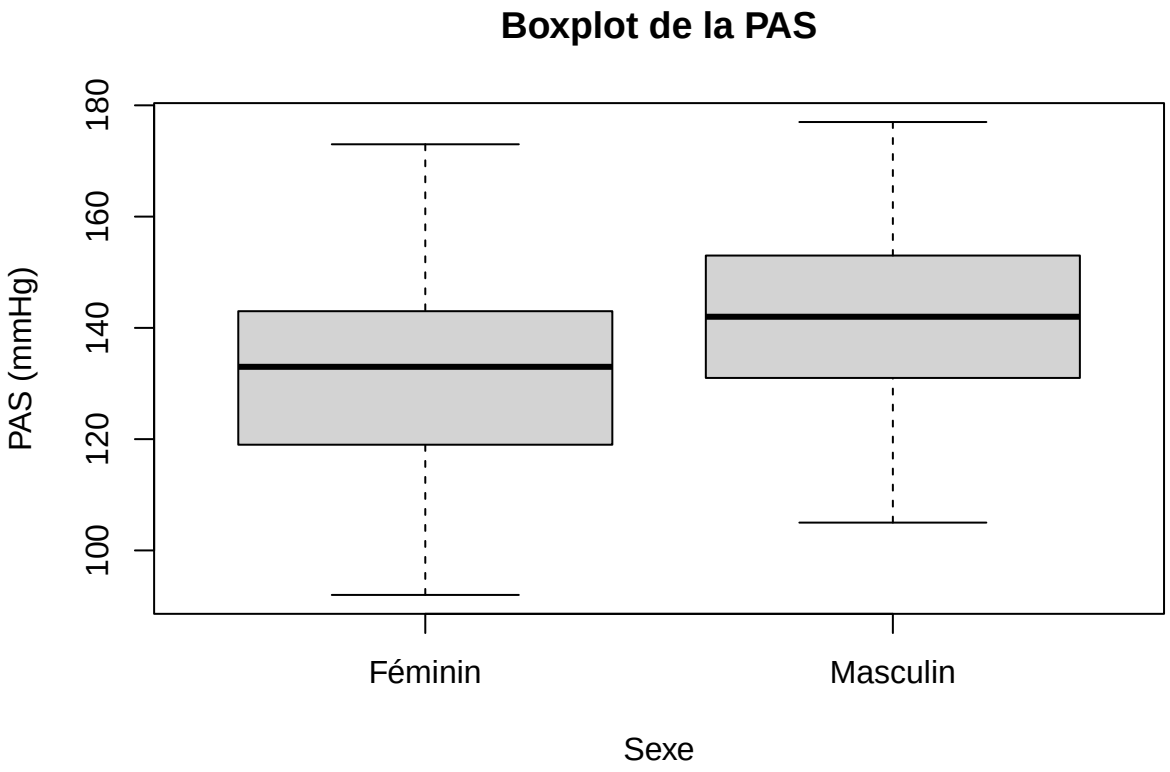


3.6.2 Distributions bivariées

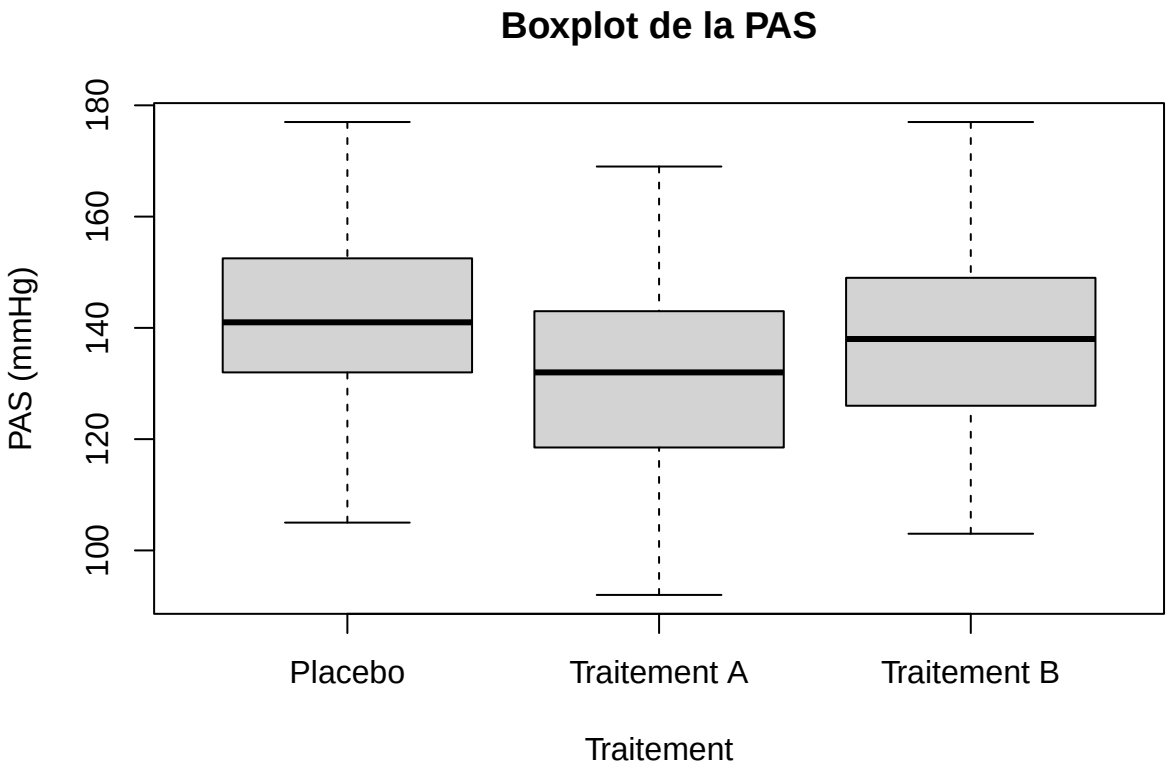
On peut croiser une variable quantitative en fonction des modalités d'une variable qualitative à l'aide de box-plots.

Une bonne pratique est d'indiquer clairement le noms des axes, les titres et légendes. Pour cela, nous pouvons utiliser la base de méta-données.

```
### Le signe "tilde" ~ est souvent utilisé pour définir une variable comme une
### fonction d'une ou plusieurs autres variables :
### y = f(x1, x2, x3) s'écrit y ~ x1 + x2 + x3

### PAS en fonction du sex.
boxplot(df_1$pas ~ df_1$sexL,
        ylab = "PAS (mmHg)",
        xlab = meta_df_1$label[meta_df_1$var == "sex" & meta_df_1$id_labs == 1],
        main = "Boxplot de la PAS")
```




```r
### PAS en fonction du traitement
boxplot(df_1$pas ~ df_1$traitL,
        ylab = "PAS (mmHg)",
        xlab = meta_df_1$label[meta_df_1$var == "trait" & meta_df_1$id_labs == 1],
        main = "Boxplot de la PAS")
```



Un nuage de points peut être utilisé pour représenter le croisement de deux variables quantitatives.

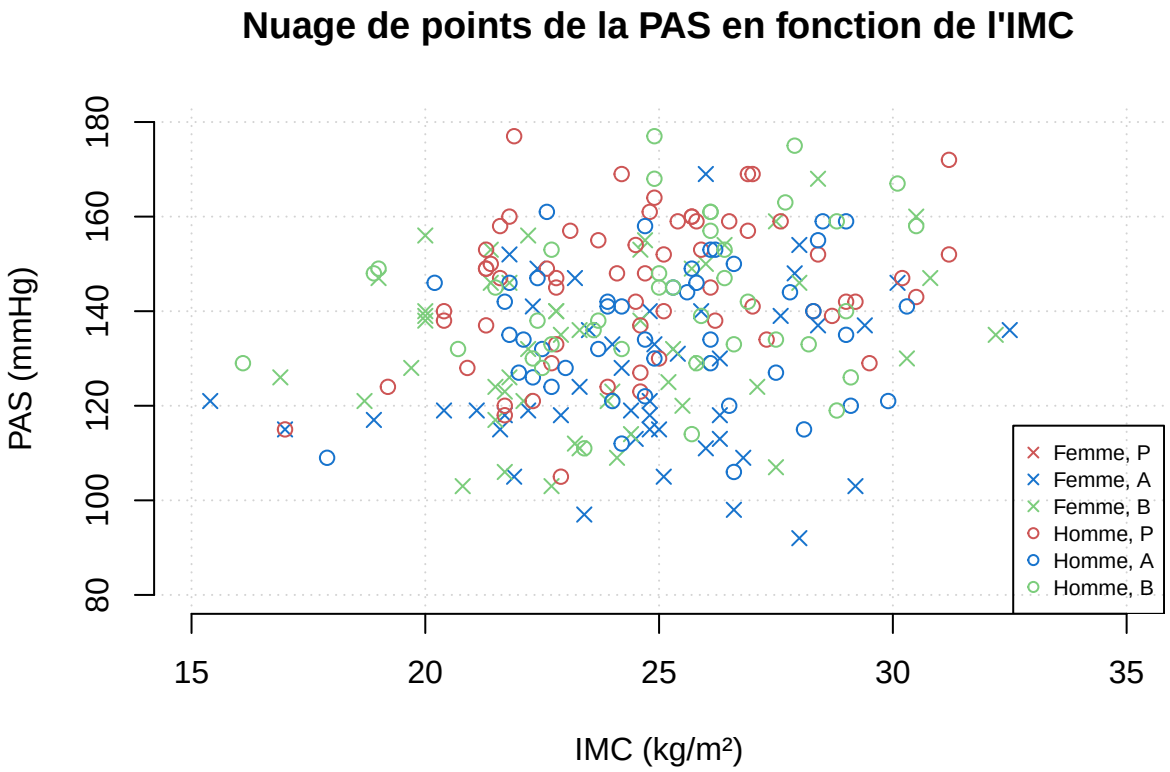
R offre beaucoup de possibilités pour modifier les paramètres graphiques. L'aide `?par` indique l'ensemble des options possibles. Il difficile d'y voir clair dans cette multitude d'options, les paramètres les plus utiles sont notamment :

- `col` : pour spécifier la couleur des points et des lignes. Ce paramètre se décline à la couleur des axes `col.axis`, des labels `col.lab`, des titres `col.main`, des sous-titres `col.sub`, …
- `pch` : pour définir le symbole des points dans les nuages de points (0 pour un carré, 1 pour un rond, 2 pour un triange, etc). Cf. le détail de l'aide de `?points()`
- `lty` : pour spécifier le type de ligne : 0 = *blank*, 1 = *solid*, 2 = *dashed*, etc. On peut également l'indiquer en caractère "blank", "solid", "dashed", …. cf
- `lwd` : pour spécifier la largeur d'une ligne
- `cex` : valeur numérique indiquant la taille relative de la police de caractères. Elle se décline pour la police utilisée sur les axes `cex.axis`, les labels `cex.lab`, les titres `cex.main`, les sous-titres `cex.sub`, etc. Ce paramètre va également influencer la taille des points dans un nuage de points
- `mfrom` et `mfcol` pour combiner plusieurs graphiques sur une ou plusieurs lignes et une ou plusieurs colonnes
- `mar` : indique le nombre de ligne par marge, indiqué dans cet ordre `c(bottom, left, top, right)`. La valeur par défaut est `c(5, 4, 4, 2) + 0.1`. `mai` : permet également de préciser la taille des marges (en pouces), dans le même ordre `c(bottom, left, top, right)`
- `legend` : permet d'ajouter une légende
- etc.

La fonction exemple permet d'avoir des exemples d'utilisation paramètres graphiques : `example(mar)`, `example(line)`, `example(axis)`, `example(legend)`

```
### On peut représenter un nuage de points de la PAS en fonction de l'IMC :
### où les hommes et les femmes ont deux symboles différents et les traitements
### deux couleurs différentes
par(mfrow = c(1, 1)) # pour revenir à un seul graphique par sortie
```

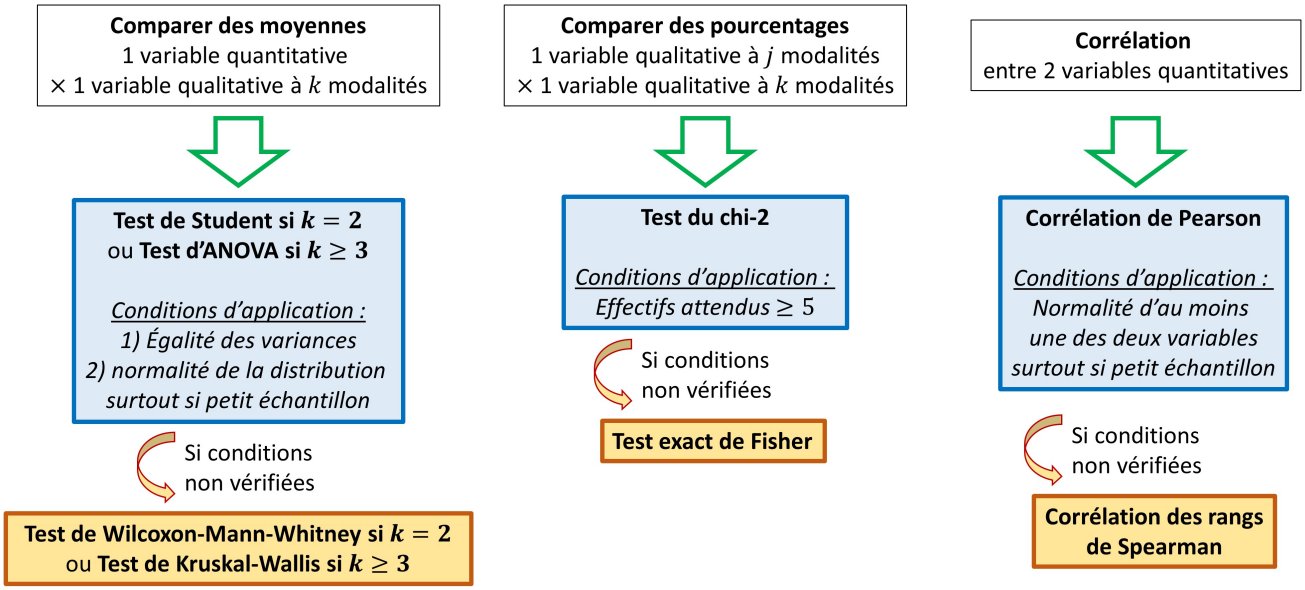
```r
plot.new()
par(mar = c(5,4,4,2) + 0.1) # paramètre des marges par défaut
plot.window(xlim = c(15, 35), # range(df_1$imc)
            ylim = c(80, 180)) # range(df_1$pas)
grid() # ajoute une grille
### ajoute le nuage de points des femmes, groupe placebo
points(data = subset(df_1, subset = c(sex == "Féminin" & trait == "Placebo")),
       pas ~ imc,
       col = "indianred3", # placebo en rouge
       pch = 4) # femme avec une croix
# ajoute le nuage de points des femmes, groupe traitement A
points(data = subset(df_1, subset = c(sex == 0 & trait == 2)),
       pas ~ imc,
       col = "dodgerblue3", # traitement A en bleu
       pch = 4) # femme avec une croix
# ajoute le nuage de points des femmes, groupe traitement B
points(data = subset(df_1, subset = c(sex == 0 & trait == 3)),
       pas ~ imc,
       col = "palegreen3", # traitement B en vert
       pch = 4) # femme avec une croix
# ajoute le nuage de points des hommes, groupe placebo
points(data = subset(df_1, subset = c(sex == 1 & trait == 1)),
       pas ~ imc,
       col = "indianred3", # placebo en rouge
       pch = 1) # hommes avec un rond
# ajoute le nuage de points des hommes, groupe traitement A
points(data = subset(df_1, subset = c(sex == 1 & trait == 2)),
       pas ~ imc,
       col = "dodgerblue3", # traitement A en bleu
       pch = 1) # hommes avec un rond
# ajoute le nuage de points des hommes, groupe traitement B
points(data = subset(df_1, subset = c(sex == 1 & trait == 3)),
       pas ~ imc,
       col = "palegreen3", # traitement B en vert
       pch = 1) # hommes avec un rond
axis(1, # axe du bas
     lwd = 1, # largeur de la ligne
     font.axis=1) # taille de la police de caractère
axis(2, # axe à gauche
     lwd = 1, # largeur de la ligne
     font.axis=1) # taille de la police de caractère
title(xlab = "IMC (kg/m²)")
title(ylab = "PAS (mmHg)")
title(main = "Nuage de points de la PAS en fonction de l'IMC")
legend("bottomright",
       c("Femme, P", "Femme, A", "Femme, B",
         "Homme, P", "Homme, A", "Homme, B"),
       pch = c(4,4,4,1,1,1),
       col = c("indianred3", "dodgerblue3", "palegreen3",
               "indianred3", "dodgerblue3", "palegreen3"),
       ncol = 1,
       cex = 0.7)
```



Pour trouver des informations détaillées sur l'utilisation des fonction graphiques de R base, vous pouvez lire le tutoriel détaillé de Karolis Koncevičius.

3.7 Analyses bivariées

Normalement, vous devriez savoir choisir quel test de comparaison utiliser en fonction des variables à comparer.



3.7.1 Variable quantitative × qualitative

Pour décrire une variable quantitative en fonction d'une variable qualitative, on peut utiliser les fonctions `aggregate()` et `tapply()`.

La fonction `aggregate()` permet d'appliquer des fonctions (mean(), sd(), etc) à des sous-groupes de variables définies en facteurs.

```
### Description de la moyenne de l'IMC et de la PAS en fonction du facteur sex
aggregate(x = df_1[,c("imc", "pas")], # variable(s) quantitative(s)
          by = list(df_1$sexL), # en fonction du "factor" sexL
          FUN = mean) # fonction à appliquer à la variable x
```

```
##    Group.1      imc      pas
## 1  Féminin 24.09281 132.4248
## 2 Masculin 24.88503 141.9932
```

Pour décrire de manière synthétique, la distribution, on veut récupérer l'effectif, la moyenne et l'écart-type par groupe. Nous allons modifier la fonction "maison" `univ_quanti()` que nous avions défini précédemment en ajoutant un argument `details` :

- si `details = TRUE`, alors la fonction retourne l'ensemble des résultats détaillés (effectifs, moyenne, écart-type, minimum, 1er quartile, médiane, 3ème quartile et maximum)
- si `details = FALSE`, alors la fonction retourne uniquement les effectifs, la moyenne et l'écart-type.

Pour cela nous allons compléter la fonction avec la construction `if(condition) {expression1} else {expression2}` : l'expression1 est appliquée si la condition est vraie, sinon l'expression2 est appliquée.

```r
univ_quanti <- function(x, dig = 2, remove_miss = TRUE, details = TRUE) {
  n <- length(x[!is.na(x)])
  moy <- mean(x, na.rm = remove_miss)
  sd <- sd(x, na.rm = remove_miss)
  q <- quantile(x, probs = c(0, 0.25, 0.5, 0.75, 1), na.rm = remove_miss)

  # construction if (condition) {expression1} else {expression2}
  if (details == TRUE) { # si la condition entre parenthèse est vraie ...
    param <- c(n,        # ... appliquer les fonctions entre accolades
               round(moy, digits = dig),
               round(sd, digits = dig),
               q)
  } else { # sinon appliquer les fonctions entre la 2ème accolade
    param <- c(n,
               round(moy, digits = dig),
               round(sd, digits = dig))
  }

  # ajouter un nom à chaque élément du vecteur
  if (details == TRUE) {
    names(param) <- c("N", "mean", "sd", "min", "Q1", "median", "Q3", "max")
  } else {
    names(param) <- c("N", "mean", "sd")
  }

  # retourne les résultats
  return(param)
}

### Description de la PAS en fonction du sexe (facteur sexL)
aggregate(x = df_1$pas,
          by = list(df_1$sexL),
          FUN = univ_quanti, # fonction à utiliser
          dig = 1, remove_miss = TRUE, details = FALSE)
```

```
##    Group.1   x.N x.mean  x.sd
## 1  Féminin 153.0  132.4  16.8
## 2 Masculin 147.0  142.0  15.5
```

```r
### Description de la PAS en fonction du traitement (facteur traitL)
aggregate(x = df_1$pas,
          by = list(df_1$traitL),
          FUN = univ_quanti, # fonction à utiliser
          dig = 1, remove_miss = TRUE, details = FALSE)
```

```
##        Group.1   x.N x.mean  x.sd
## 1      Placebo 120.0  141.4  15.6
## 2 Traitement A  91.0  130.7  16.4
## 3 Traitement B  89.0  137.8  17.0
```

La fonction `tapply()` permet également d'appliquer une fonction selon les sous-groupes d'un facteur (indiqué en argument `INDEX`). Les résultats obtenus sont au format de liste.

```
### Description de l'IMC en fonction du sexe (facteur sexL)
tapply(X = df_1$imc,
       INDEX = list(df_1$sexL), # facteur à indiquer sous forme de liste.
       FUN = univ_quanti, # fonction à utiliser sur la variable X
       dig = 1, remove_miss = TRUE, details = FALSE)
```

```
## $Féminin
##     N  mean    sd
## 153.0  24.1   3.1
##
## $Masculin
##     N  mean    sd
## 147.0  24.9   3.0
```

```
### Description de l'IMC en fonction du sexe (facteur sexL)
tapply(X = df_1$imc,
       INDEX = list(df_1$traitL), # facteur à indiquer sous forme de liste.
       FUN = univ_quanti, # fonction à utiliser sur la variable X
       dig = 1, remove_miss = TRUE, details = FALSE)
```

```
## $Placebo
##     N  mean    sd
## 120.0  24.3   2.9
##
## $`Traitement A`
##    N mean   sd
## 91.0 24.8  3.0
##
## $`Traitement B`
##    N mean   sd
## 89.0 24.3  3.3
```

3.7.2 Variable qualitative × qualitative

On peut décrire une variable qualitative en fonction d'une autre variable qualitative avec les fonctions `table()` et `prop.table(table())`.

```
### Décrire les effectifs de la variable sexL (en lignes),
### en fonction de la variable traitL (en colonnes) avec la fonction table()
### on va stocker les résultats dans un objet "sex_by_trait_N"
sex_by_trait_N <- table(df_1$sexL, df_1$traitL)
sex_by_trait_N
```

```
##
##            Placebo Traitement A Traitement B
##   Féminin       57           46           50
##   Masculin      63           45           39
```

```
### Les pourcentages peuvent être décrits avec la fonction prop.table()
### il faut préciser l'argument margin = 1 pour des pourcentages en ligne,
### ou margin = 2 pour des pourcentages en colonnes
### ou margin = NULL pour des pourcentages par cellule.
### on va stocker les pourcentage dans un objet "sex_by_trait_pct"
```

```
sex_by_trait_pct <- prop.table(sex_by_trait_N,
                               margin = 2) #  % par colonne
sex_by_trait_pct
```

```
##
##              Placebo Traitement A Traitement B
##   Féminin  0.4750000    0.5054945    0.5617978
##   Masculin 0.5250000    0.4945055    0.4382022
```

```
### On peut combiner ces deux résultats avec la fonction paste0() qui combine
### chaque élément de 2 vecteurs
tab_biv_quali <- paste0(sex_by_trait_N,
                        "(", round(sex_by_trait_pct * 100, digits = 1), "%)")
tab_biv_quali
```

```
## [1] "57(47.5%)" "63(52.5%)" "46(50.5%)" "45(49.5%)" "50(56.2%)" "39(43.8%)"
```

```
### c'est devenu un vecteur atomique de caractères
### on va lui redonner les dimensions et noms des matrices initiales
### en lui redonnant les attribut dim() et dimnames() des tables initiales
dim(tab_biv_quali) <- dim(sex_by_trait_N)
dimnames(tab_biv_quali) <- dimnames(sex_by_trait_N)
tab_biv_quali
```

```
##
##            Placebo     Traitement A Traitement B
##   Féminin  "57(47.5%)" "46(50.5%)"  "50(56.2%)"
##   Masculin "63(52.5%)" "45(49.5%)"  "39(43.8%)"
```

3.7.3 Comparer 2 moyennes

On peut **comparer deux moyennes avec le test de Student** `t.test()`.

```
### Comparaion de la moyenne de PAS en fonction du sexe (2 moyennes)
### par un test de Student
ttest_pas_sex <- t.test(data = df_1, # préciser quelle est la base de donnée
                        pas ~ sexL, # moyenne de la PAS en fonction du sexe
                        var.equal = TRUE)
## Attention, par défaut, l'argument var.equal = FALSE
# il considère que l'hypothèse d'égalité des variances est fausse et applique
# un test de Welch qui est robuste, même en cas de variances inégales)
## Pour réaliser le test de Student, il faut indiquer : var.equal = TRUE
ttest_pas_sex
```

```
##
##  Two Sample t-test
##
## data:  pas by sexL
## t = -5.1297, df = 298, p-value = 5.239e-07
## alternative hypothesis: true difference in means between group Féminin and group Masculin is not equal
## 95 percent confidence interval:
##  -13.239193  -5.897528
## sample estimates:
##  mean in group Féminin mean in group Masculin
##               132.4248               141.9932
```

```
## Les résultats indiquent que l'on peut rejeter l'hypothèse nulle d'égalité des
## moyennes entre les hommes et les femmes,
## de manière statistiquement significative, avec une p-value = 5.2e-07

## On peut récupérer des éléments spécifiques de la liste de résultats avec
## l'opérateur dollar appliqué à l'objet où on stocke les résultats :
ttest_pas_sex$p.value # pour récupérer uniquement la p-value
```

```
## [1] 5.238798e-07
```

```
ttest_pas_sex$conf.int # pour l'intervalle de confiance à 95% de la différence
```

```
## [1] -13.239193  -5.897528
## attr(,"conf.level")
## [1] 0.95
```

Les **conditions d'application du test de Student** sont :

- l'égalité des variances
- et la normalité de la distribution dans chaque groupe (surtout si les effectifs sont faibles). Dans notre exemple avec un effectif de 300, les résultats seront robuste même en cas d'écart à la normalité.

Pour vérifier l'égalité des variance, on peut utiliser le test de Levene `leveneTest()` qui est disponible dans le package `car` qui n'est pas chargé en mémoire. Pour pouvoir utiliser le test de Levene, il faut :

- soit commencer par charger le package avec `library(car)`, puis lancer la fonction `leveneTest()`
- soit utiliser la syntaxe `car::leveneTest()` qui permet de lancer uniquement la fonction `leveneTest()` sans charger la totalité du package `car`.

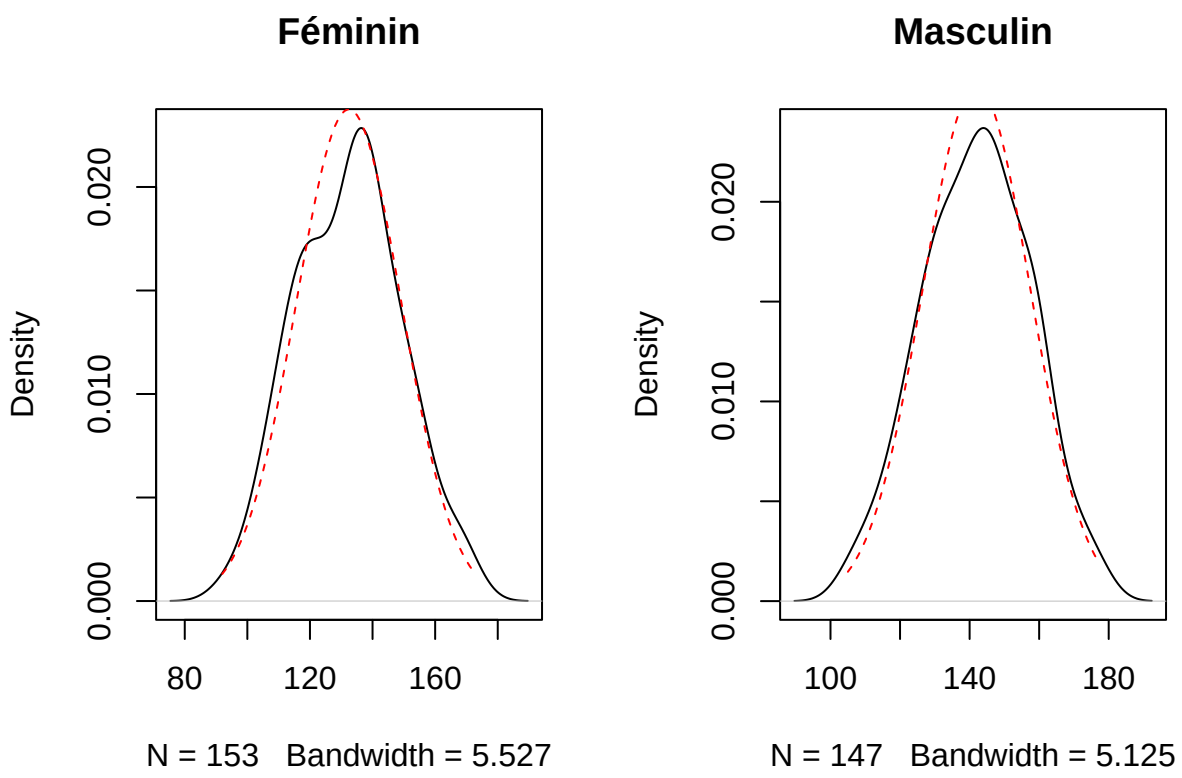
```
# note : la variable en classe (sexe) doit être de type "factor"
car::leveneTest(data = df_1, pas ~ sexL)
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value Pr(>F)
## group   1  1.0178 0.3139
##       298
```

La p-value est égale à 0.31 : on ne rejette pas l'hypothèse nulle d'égalité des variances (par abus d'interprétation, on va considérer que la condition d'égalité des variances est vraie).

Pour vérifier la normalité de la distribution de la PAS par sexe, on peut tracer la densité de kernel dans chaque sous-groupe. Voici un exemple de programmation (qui ajoute une courbe normale en pointillés rouge pour servir de référence) :

```
par(mfrow = c(1, 2)) # les 2 figures seront présentées sur 1 ligne
for (i in 1:2) {
  vect <- df_1$pas[as.integer(df_1$sexL) == i]
  plot(density(vect), # densité de kernel
       main = attributes(df_1$sexL)$levels[i])
  lines(x = seq(min(vect), max(vect), length(50)), # loi normale de référence
        y = dnorm(seq(min(vect), max(vect), length(50)),
                  mean = mean(vect),
                  sd = sd(vect)),
        col = "red", lty = "dashed")
}
```




```r
rm(vect) # supprime l'objet "vect" qui n'est plus utile
```

Ici, les distributions semblent assez proches de lois normales.

Si les conditions d'application du test de Student ne sont pas vérifiées, on peut utiliser le test des rangs de Wilcoxon-Mann-Whitney `wilcox.test()`).

```r
wilcox.test(data = df_1, pas ~ sex) # p-value = 1.529e-06
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  pas by sex
## W = 7609.5, p-value = 1.29e-06
## alternative hypothesis: true location shift is not equal to 0
```

Le test de Wilcoxon rejette l'hypothèse nulle d'égalité des moyennes de PAS entre les hommes et les femmes (de manière significative, avec une p-value = 1.5e-06).

3.7.4 Comparer 3 moyennes ou plus

On peut **comparer 3 moyennes ou plus avec une Anova** `anova()`. Par exemple, si on compare la moyenne de PAS en fonction du traitement, l'hypothèse nulle est : *"il n'existe pas de différence de moyenne de PAS entre les 3 groupes de traitement"*.

Dans R, la fonction `anova()` s'applique au résultat d'un modèle linéaire. Nous allons donc d'abord estimer le modèle linéaire de la pression artérielle systolique en fonction du traitement avec la fonction `lm()` (pour *linear model*). Nous

pourrons également utiliser les résidus de ce modèle pour vérifier la condition de normalité. La condition d'égalité des variances pourra être vérifiée comme précédemment avec le test de Levenne.

```
mod_pas_trait <- lm(pas ~ traitL, # modèle de PAS en fonction du traitement
                    data = df_1)
# le fait d'utiliser un "factor" (traitL) comme variable explicative
# va permettre à R de créer automatiquement deux indicatrices pour modéliser
# le traitement à 3 catégories, en prenant la première catégorie ("placebo")
# comme référence
summary(mod_pas_trait)
```

```
##
## Call:
## lm(formula = pas ~ traitL, data = df_1)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -38.747 -11.747   0.157  11.340  39.157
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)          141.400      1.486  95.182  < 2e-16 ***
## traitLTraitement A   -10.653      2.262  -4.709 3.82e-06 ***
## traitLTraitement B    -3.557      2.277  -1.563    0.119
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 16.27 on 297 degrees of freedom
## Multiple R-squared:  0.07022,    Adjusted R-squared:  0.06396
## F-statistic: 11.22 on 2 and 297 DF,  p-value: 2.016e-05
```

Le modèle linéaire indique que :

- la PAS est en moyenne de 141.4 mmHg dans le groupe de référence (placebo) avec l'intercept.
- la PAS est plus faible de -10.7 mmHg en moyenne dans le groupe "traitement A" par rapport au placebo (de manière significative, p = 3.8e-06)
- la PAS est plus faible de -3.6 mmHg en moyenne dans le groupe "traitement B" par rapport au placebo (de manière non significative, p = 0.12)

```
## On applique ensuite ce modèle linéaire à la fonction anova
## qui va retourner une table d'Anova avec les différentes sources de
## variabilité, degrés de liberté, sommes des carrés, carrés moyens,
## statistique F du test de Fisher et p-value.
anova_pas_trait <- anova(mod_pas_trait)
anova_pas_trait
```

```
## Analysis of Variance Table
##
## Response: pas
##            Df Sum Sq Mean Sq F value    Pr(>F)
## traitL      2   5940 2970.18  11.215 2.016e-05 ***
## Residuals 297  78656  264.83
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
## on peut récupérer des résultats spécifiques avec l'opérateur dollar
anova_pas_trait$`Pr(>F)` # récupère uniquement la p-value
```

```
## [1] 2.015778e-05           NA
```

On peut vérifier la condition d'égalité des variances avec un test de Levenne :

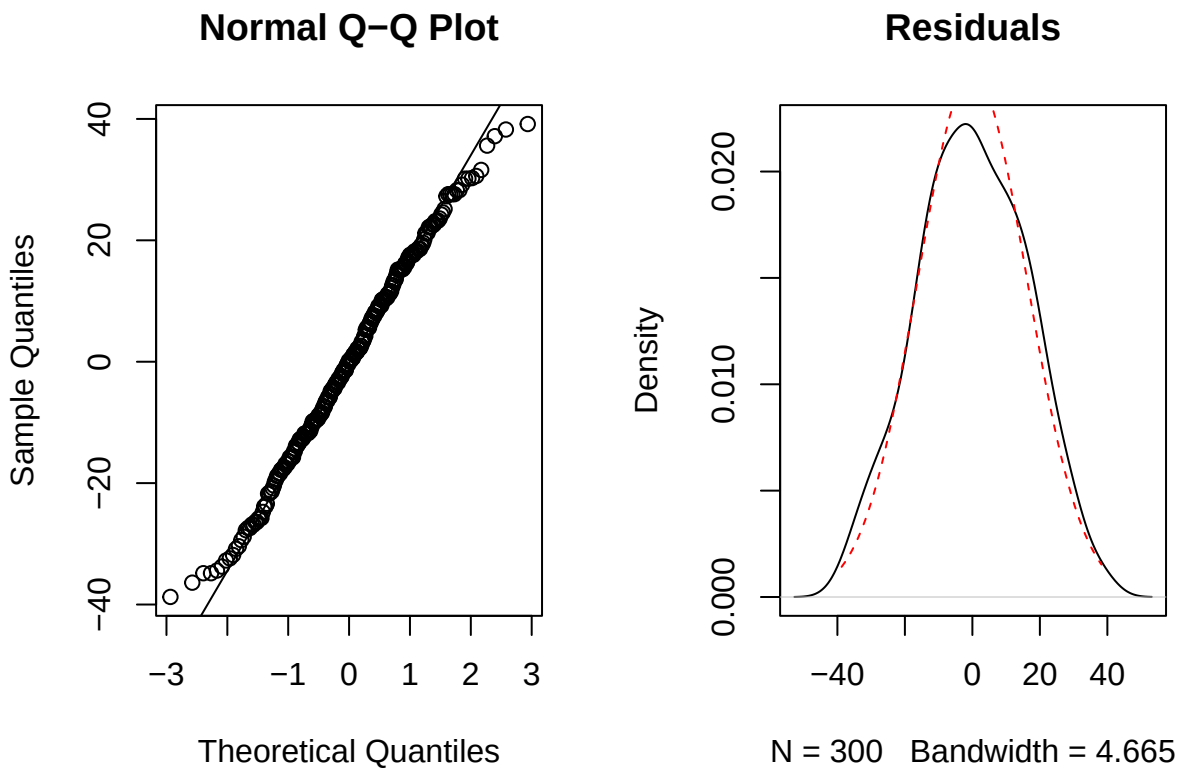
```
car::leveneTest(data = df_1, pas ~ traitL) # p = 0.4703   OK
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value Pr(>F)
## group   2  0.6669  0.514
##       297
```

On ne rejette pas l'hypothèse nulle d'égalité des variances, car la p-value n'est pas significative (p = 0.51). Par abus d'interprétation, on considère que les variances sont égales.

Pour vérifier la condition de normalité, on va évaluer la normalité des résidus du modèle linéaire.

```
par(mfrow = c(1, 2)) # 2 graphiques sur 1 ligne

### On peut récupérer les résidus du modèle linéaire avec l'opérateur dollar
### Représenter les résidus dans un QQ-plot
qqnorm(mod_pas_trait$residuals)
qqline(mod_pas_trait$residuals)

### Représenter la distribution des résidus par une densité de kernel
vect <- mod_pas_trait$residuals
plot(density(vect), # densité de kernel
     main = "Residuals")
lines(x = seq(min(vect), max(vect), length(50)), # loi normale de référence
      y = dnorm(seq(min(vect), max(vect), length(50)),
                mean = mean(vect),
                sd = sd(vect)),
      col = "red", lty = "dashed")
```




```
rm(vect)
```

La condition normalité semble acceptable.

Si les variances avaient été inégales, on aurait appliqué un test de Kruskal-Wallis, avec la fonction `kruskal.test()`.

```
kruskal.test(pas ~ traitL, data = df_1) # p-value = 7.336e-05
```

```
##
##  Kruskal-Wallis rank sum test
##
## data:  pas by traitL
## Kruskal-Wallis chi-squared = 19.04, df = 2, p-value = 7.336e-05
```

Le test de Kruskal-Wallis rejette l'hypothèse nulle d'égalité des moyenne entre les 3 groupes de traitement, de manière significative (p = 7.3e-05).

3.7.5 Comparer des pourcentages

On peut **comparer plusieurs pourcentages avec un test du chi-2** `chisq.test()`.

Par exemple, nous allons tester l'hypothèse nulle suivante : _"H0 : La répartition par sexe est la même dans les 3 groupes de traitement_"

Pour répondre à la question, nous pouvons faire un test comparant le pourcentage d'hommes et de femmes au sein des 3 groupes de traitement. Nous avons déjà vu que les fonction `table()` et `prop.table(table())` permettaient de décrire la répartition du traitement en fonction du sexe.

La fonction `chisq.test()` va s'appliquer à un tableau croisé des effectifs par traiement et par sexe. On rappelle que les conditions d'application à vérifier sont que tous les effectifs attendus doivent être ≥ 5.

```
### Appliquer un test du chi2 au tableau croisé du traitement en fonction du sex
# Attention, par défaut la fonction chisq.test applique la correction de Yates
# en indiquant correct = FALSE
# cette correction n'est utile que si les conditions d'application du test
# du chi-2 ne sont pas vérifiées.
chi2 <- chisq.test(table(df_1$sexL, df_1$traitL),
                   correct = FALSE)
chi2
```

```
##
##  Pearson's Chi-squared test
##
## data:  table(df_1$sexL, df_1$traitL)
## X-squared = 1.5512, df = 2, p-value = 0.4604
```

D'après ces résultats, on ne rejette pas l'hypothèse nulle d'égalité de la répartition par sexe dans les 3 groupes de traitement (p = 0.46)

Pour récupérer la table des effectifs attendus et vérifier les conditions d'application, on peut utiliser l'opérateur dollar `$expected` :

```
chi2$expected
```

```
##
##            Placebo Traitement A Traitement B
##   Féminin     61.2        46.41        45.39
##   Masculin    58.8        44.59        43.61
```

On voit que la condition d'application est vérifiée car tous les effectifs attendus sont ≥ 5.

Si les conditions n'avaient pas été vérifiées, on aurait pu appliquer un test exact de Fisher `fisher.test()`

```
fisher.test(table(df_1$sexL, df_1$traitL))
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  table(df_1$sexL, df_1$traitL)
## p-value = 0.4536
## alternative hypothesis: two.sided
```

D'après le test exact de Fisher, on ne peut pas rejetter l'hypothèse nulle d'égalité de la répartition par sexe dans les 3 groupes de traitement.

3.7.6 Corrélations

On peut calculer les **corrélations de Pearson et de Spearman** entre la PAS et l'IMC avec la fonction `cor()`. La fonction `cor.test()` teste l'hypothèse nulle $H0 : \rho = 0$ (elle recalcule également le coefficient de corrélation ainsi qu'un intervalle de confiance). Au sein de ces deux fonctions, vous pouvez préciser si vous souhaiter calculer et tester une corrélation de Pearson ou de Spearman avec l'argument `method`.

```r
### Estimer puis tester une corrélation de Pearson entre PAS et IMC
rho_pearson <- cor.test(df_1$imc, df_1$pas, method = "pearson")
rho_pearson
```

```
##
##  Pearson's product-moment correlation
##
## data:  df_1$imc and df_1$pas
## t = 3.8534, df = 298, p-value = 0.0001427
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.1072647 0.3231284
## sample estimates:
##       cor
## 0.2178593
```

Le coefficient de corrélation de Pearson est $\rho_1 = 0.218$, il est signficativement différent de 0 (p=0.00014).

```r
### Estimer puis tester une corrélation de Pearson entre PAS et IMC
rho_spearm <- cor.test(df_1$imc, df_1$pas, method = "spearman")
```

```
## Warning in cor.test.default(df_1$imc, df_1$pas, method = "spearman"):
## Impossible de calculer la p-value exacte avec des ex-aequos
```

```r
rho_spearm
```

```
##
##  Spearman's rank correlation rho
##
## data:  df_1$imc and df_1$pas
## S = 3522826, p-value = 0.0001503
## alternative hypothesis: true rho is not equal to 0
## sample estimates:
##      rho
## 0.217141
```

Le coefficient de corrélation de Spearmon est $\rho_2 = 0.217$, il est signficativement différent de 0 (p=0.00015).

3.8 Analyse multivariée

Nous allons estimer un modèle multivarié de la moyenne de pression artérielle systolique, en fonction du traitement, ajusté sur le sexe et l'IMC.

Pour cela nous allons utiliser la fonction `lm()` (*linear model*) :

```r
model <- lm(pas ~ traitL + sexL + imc,
            data = df_1)
summary(model)
```

```
##
## Call:
## lm(formula = pas ~ traitL + sexL + imc, data = df_1)
```

```
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -38.871 -10.083  -0.342  10.941  41.171
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)         109.5668     7.1403  15.345  < 2e-16 ***
## traitLTraitement A -10.9623     2.1246  -5.160 4.55e-07 ***
## traitLTraitement B  -2.8373     2.1374  -1.327 0.185388
## sexLMasculin         8.5634     1.7796   4.812 2.39e-06 ***
## imc                  1.1240     0.2904   3.870 0.000134 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.24 on 295 degrees of freedom
## Multiple R-squared:  0.1902, Adjusted R-squared:  0.1792
## F-statistic: 17.32 on 4 and 295 DF,  p-value: 8.911e-13
```

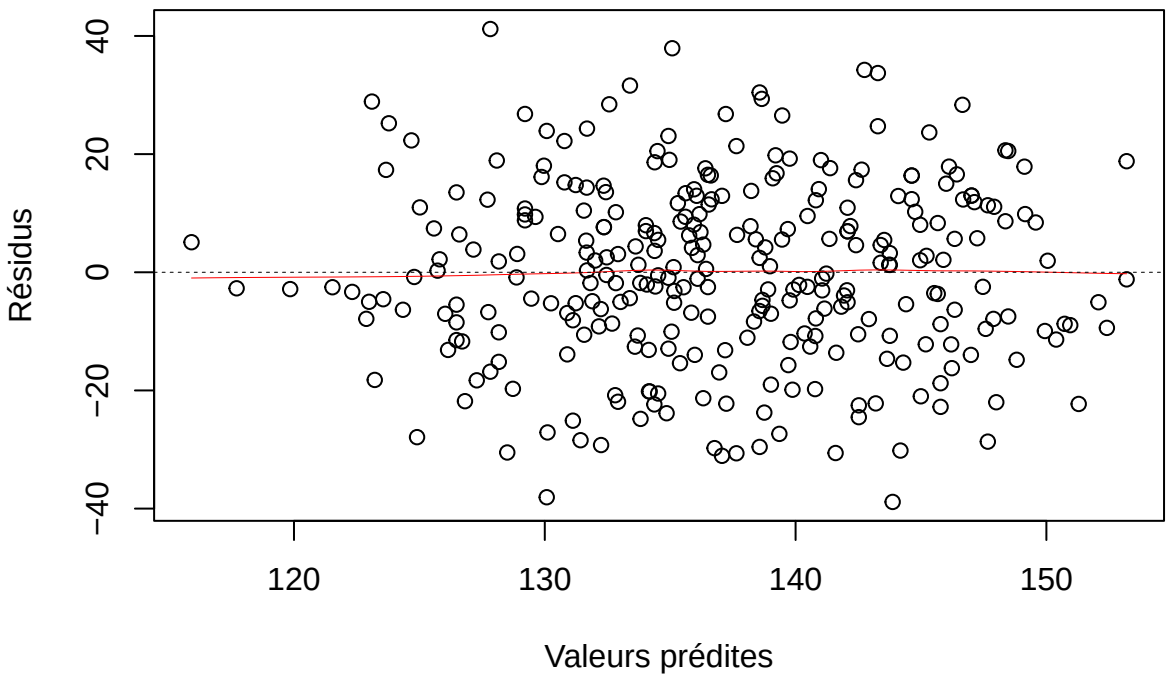
Après ajustement sur le sexe et l'IMC, on observe que :

- la PAS était en moyenne inférieure de -11.0 mmHg dans le groupe traitement A par rapport au groupe placebo, de manière significative (p=4.6e-07),
- la PAS était en moyenne inférieure de -2.8 mmHg dans le groupe traitement B par rapport au groupe placebo, de manière non-significative (p=0.19).

Pour vérifier les conditions d'application, la méthode la plus classique est de représenter le nuage de points des résidus en fonction des valeurs prédites par le modèle. On peut se servir de l'opérateur dollar pour récupérer :

- les résidus du modèle `model$residuals`
- les valeurs prédites par le modèle `model$fitted.values`

```r
par(mfrow = c(1, 1)) # 1 figure sur 1 ligne et 1 colonne

# la fonction suivant trace une courbe lissée (loess)
# au milieu d'un nuage de points
scatter.smooth(model$fitted.values, model$residuals,
               lpars = list(col = "red", lwd = 0.5, lty = 1),
               xlab = "Valeurs prédites", ylab = "Résidus")
abline(h = 0, # ajoute une ligne horizontale en 0
       lwd = 0.5, # largeur de ligne (line width)
       lty = 2) # type de ligne = pointillé
```



De manière encore plus automatisée, la commande `plot(model)` permet d'obtenir une série de 4 graphiques de diagnostic post-estimation d'un modèle de régression linéaire (tapez la touche "entrée" 4 fois pour obtenir les graphiques) :

- le 1er graphique est le nuage de points des résidus en fonction des valeurs prédites
- le 2ème graphique permet de vérifier la normalité des résidus standardisés
- le 3ème graphique est le nuage de points de la racine carrée des résidus standardisés en fonction des valeurs prédites
- le 4ème graphique le nuage de points des résidus standardisés en fonction des distances de Cook (pour évaluer l'effet levier)

Pour rappel, vous pouvez sauvegarder le résultat de ce modèle linéaire sous forme d'objet R avec tous ses attributs en utilisant la fonction `saveRDS()`. Cela vous permet de stocker, puis recharger ces résultats directement dans R.

```
### Créez un dossier "results" dans votre dossier de travail
### soit directement dans l'environnement windows ou macOS ou linux,
### soit avec la commande dir.create()
dir.create("results")

### Sauvegardez l'objet contenant le modèle linéaire
saveRDS(model, "results/model.R")

### si vous videz la totalité des objets présents dans l'environnement
rm(list = ls())

### Vous pourrez re-importez le modèle original avec readRDS(),
### avec tous les attributs contenus dans le modèle
model <- readRDS("results/model.R")
```

```r
summary(model)
# Call:
# lm(formula = pas ~ traitL + sexL + imc, data = df_1)
#
# Residuals:
#     Min      1Q  Median      3Q     Max
# -38.871 -10.083  -0.342  10.941  41.171
#
# Coefficients:
#                     Estimate Std. Error t value Pr(>|t|)
# (Intercept)         109.5668     7.1403  15.345  < 2e-16 ***
# traitLTraitement A -10.9623     2.1246  -5.160 4.55e-07 ***
# traitLTraitement B  -2.8373     2.1374  -1.327 0.185388
# sexLMasculin         8.5634     1.7796   4.812 2.39e-06 ***
# imc                  1.1240     0.2904   3.870 0.000134 ***
# ---
# Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
#
# Residual standard error: 15.24 on 295 degrees of freedom
# Multiple R-squared:  0.1902,  Adjusted R-squared:  0.1792
# F-statistic: 17.32 on 4 and 295 DF,  p-value: 8.911e-13

qqnorm(model$residuals)

scatter.smooth(model$fitted.values, model$residuals,
               lpars = list(col = "red", lwd = 0.5, lty = 1),
               xlab = "Valeurs prédites", ylab = "Résidus")
abline(h = 0, # ajoute une ligne horizontale en 0
       lwd = 0.5, # largeur de ligne (line width)
       lty = 2) # type de ligne = pointillé
```

Il n’est pas nécessaire de relancer toute l’analyse pour retrouver ces résultats !

Chapter 4

Pipe natif, `within()` et `with()`

La syntaxe de R base peut être rendue plus lisible avec le pipe natif de R `|>`, ainsi que les fonctions `within()` et `with()`.

Nous allons reprendre les éléments de l'analyse réalisée au chapitre 3 en utilisant cette syntaxe complémentaire.

Commençons par supprimer l'ensemble des éléments de l'environnement, puis importons à nouveau les bases `df_1` et `meta_df_1` :

```
### Vider l'environnement
rm(list = ls())

### Importer df_1 et meta_df_1
df_1 <- read.csv2("data/df_1.csv")
meta_df_1 <- read.csv2("data/meta_df_1.csv")
```

4.1 Pipe natif de R

Le principe du "pipe" est une syntaxe de programmation qui a été initialement introduite avec le package `maggritr` au sein de la famille de packages du "tidyverse". Ce principe a été repris dans R base à partir de sa version 4.1.0.

Le pipe natif s'écrit avec une barre verticale suivi d'un signe supérieur `|>`.

La syntaxe est la suivante :

- L'élément qui se site à gauche du pipe (LHS, *left hand side*) …
- … est transmis au premier argument de la fonction à droite du pipe (RDS, *right hand side*).

`x |> funct(arguments = ...)` est équivalent à `funct(x, argument = ...)`

Par exemple :

```
df_1 |> head() # est équivalent de head(df_1)
```

```
##   subjid sex  imc trait pas
## 1      1   0 24.8     2 140
## 2      2   0 24.1     3 109
## 3      3   0 26.4     1 156
## 4      4   0 23.3     2 124
## 5      5   0 25.4     2 131
## 6      6   1 25.0     3 148
```

```
df_1 |> tail()  # est équivalent de tail(df_1)
```

```
##     subjid sex  imc trait pas
## 295    295   1 24.1     1 148
## 296    296   0 18.7     3 121
## 297    297   0 23.3     3 111
## 298    298   1 27.5     3 134
## 299    299   1 24.7     2 158
## 300    300   1 22.8     1 147
```

```
df_1 |> str() # est équivalent de str(df_1)
```

```
## 'data.frame':    300 obs. of  5 variables:
##  $ subjid: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ sex   : int  0 0 0 0 0 1 0 0 0 0 ...
##  $ imc   : num  24.8 24.1 26.4 23.3 25.4 25 25.2 21.5 21.8 25.9 ...
##  $ trait : int  2 3 1 2 2 3 3 3 1 1 ...
##  $ pas   : int  140 109 156 124 131 148 125 117 132 133 ...
```

Cela permet d'avoir un code dont la décomposition est plus lisible, en enchaînant les pipes les uns après les autres :

```
# au lieu d'écrire :
round(mean(df_1$imc, na.rm = TRUE), digits = 1)
```

```
## [1] 24.5
```

```
# on peut écrire, de manière équivalente, avec 2 pipes qui s'enchaînent :
df_1$imc |>
  mean(na.rm = TRUE) |>
  round(digits = 1)
```

```
## [1] 24.5
```

```
# df_1$imc a été pris comme premier argument de mean(x, na.rm = TRUE)
# puis le résultat a été pris comme premier argument de round(x, digits = 1)
```

Au lieu de "transférer" l'élément à gauche du pipe au premier argument de la fonction à droite du pipe, il est possible de le transférer à n'importe quel argument à droite du pipe à l'aide du "placeholder" `_` (*underscore*).

`x |> funct(y, argument = _)` est équivalent à `funct(y, argument = x)`. Par exemple :

```
paste0("la moyenne de l'IMC est ",
       round(mean(df_1$imc, na.rm = TRUE), digits = 1),
       " kg/m2")
```

```
## [1] "la moyenne de l'IMC est 24.5 kg/m2"
```

```
### on peut placer le résultat de la moyenne arrondi en 2ème position des
### arguments dans la fonction paste0() avec le placeholder "_"
df_1$imc |>
  mean(na.rm = TRUE) |>
  round(digits = 1) |>
  paste0("la moyenne de l'IMC est ",
         a = _, # argument = placeholder
         " kg/m2")
```

```
## [1] "la moyenne de l'IMC est 24.5 kg/m2"
```

4.2 Fonction `within()` : créer/modifier des variables dans une base de données

La fonction `within()` permet d'éviter d'utiliser l'opérateur dollar `$` lorsque vous souhaitez :

- créer de nouvelles variables dans une base de données,
- modifier des variables déjà existantes,
- transformer les données selon des tâches spécifiques.

La syntaxe `within(df, newvar <- ...)` va créer la variable `newvar` dans la base `df`. Pour sauvegarder cette nouvelle variable dans la base, il faudra assigner le résultat à la base souhaitée :

```
df_1 <- within(df_1,
               sexL <- factor(sex,
                              labels = meta_df_1$labs[meta_df_1$var == "sex"]))
### est équivalent :
# df_1$sexL <- factor(df_1$sex,
#                     labels = meta_df_1$labs[meta_df_1$var == "sex"])
```

On peut créer plusieurs variables en même temps, en indiquant l'ensemble des commandes entre parenthèses (*note : les variables sont incluses dans l'ordre inverse de création dans la commande*) :

```
df_1 <- within(df_1, {
  sexL <- factor(sex,
                 labels = meta_df_1$labs[meta_df_1$var == "sex"])
  traitL <- factor(trait,
                   labels = meta_df_1$labs[meta_df_1$var == "trait"])
  obesite <- ifelse(imc >= 30, 1, 0)
  imc_cl <- rep(NA, nrow(df_1))
  imc_cl[df_1$imc < 18.5] <- 1
  imc_cl[df_1$imc >= 18.5 & df_1$imc < 25] <- 2
  imc_cl[df_1$imc >= 25 & df_1$imc < 30] <- 3
  imc_cl[df_1$imc >= 30] <- 4
})

head(df_1)
```

```
##   subjid sex  imc trait pas     sexL imc_cl obesite       traitL
## 1      1   0 24.8     2 140  Féminin      2       0 Traitement A
## 2      2   0 24.1     3 109  Féminin      2       0 Traitement B
## 3      3   0 26.4     1 156  Féminin      3       0      Placebo
## 4      4   0 23.3     2 124  Féminin      2       0 Traitement A
## 5      5   0 25.4     2 131  Féminin      3       0 Traitement A
## 6      6   1 25.0     3 148 Masculin      3       0 Traitement B
```

On voit que l'opérateur dollar `$` n'a été utilisé que pour aller récupérer l'information des labels dans la base de méta-données, ainsi que pour la sélection par indexation pour créer la variable `imc_cl`.

On peut également modifier une variable (l'opérateur dollar `$` n'est utilisé que pour la sélection par indexation):

```
### Remplacer la valeur de PAS de l'individu n°137 par 123 mmHg :
df_1 <- within(df_1,
               pas[df_1$subjid == 137] <- 123)
```

4.3 Fonction `with()` : analyser des variables dans une base de données

La fonction `with()` permet d'éviter d'utiliser l'opérateur `$` lorsque l'on souhaite :

- faire des calculs statistiques appliqués aux variables d'une base de données
- faire des calculs temporaires (quand il n'y a pas besoin de modifier les données de manière permanente)

Par exemple, pour calculer la moyenne de l'IMC dans la base df_1 :

```
# la commande ...
mean(df_1$imc)
```

```
## [1] 24.481
```

```
# ... est équivalente à la commande
with(df_1, mean(imc))
```

```
## [1] 24.481
```

On peut combiner la fonction `with()` avec le pipe natif :

```
# la commande suivante ...
paste0("la moyenne de l'IMC est égale à ",
       round(mean(df_1$imc, na.rm = TRUE),
             digits = 1),
       " kg/m2")
```

```
## [1] "la moyenne de l'IMC est égale à 24.5 kg/m2"
```

```
# ... est équivalente à ...
with(df_1,
     imc |>
       mean(na.rm = TRUE) |>
       round(digits = 1) |>
       paste0("la moyenne de l'IMC est égale à ",
              a = _, # résultat positionné avec le "placeholder" underscore
              " kg/m2"))
```

```
## [1] "la moyenne de l'IMC est égale à 24.5 kg/m2"
```

4.4 Exemples d'applications

Nous pouvons utiliser ces éléments de syntaxe pour les analyses réalisées au chapitre 2.

```
### On reprend notre fonction "maison" d'analyse quantitative univariée
univ_quanti <- function(x, dig = 2, remove_miss = TRUE, details = TRUE) {
  # calculer les effectifs, la moyenne, l'écart type et les quantiles
  n <- length(x[!is.na(x)])
  moy <- mean(x, na.rm = remove_miss)
  sd <- sd(x, na.rm = remove_miss)
  q <- quantile(x, probs = c(0, 0.25, 0.5, 0.75, 1), na.rm = remove_miss)
```

```
  # stocker les résultat dans le vecteur "param"
  if (details == TRUE) {
    param <- c(n,
               round(moy, digits = dig),
               round(sd, digits = dig),
               q)
  } else {
    param <- c(n,
               round(moy, digits = dig),
               round(sd, digits = dig))
  }

  # ajouter un nom à chaque élément du vecteur
  if (details == TRUE) {
    names(param) <- c("N", "mean", "sd", "min", "Q1", "median", "Q3", "max")
  } else {
    names(param) <- c("N", "mean", "sd")
  }

  # retourne les résultats
  return(param)
}
```

4.4.1 Analyses univariées

Pour les variables quantitatives IMC et PAS :

```
with(df_1,
     # pour des analyses répétées sur plusieurs élémnets de df_1,
     # on peut les stocker dans une liste
     list(IMC = univ_quanti(imc, dig = 1, remove_miss = TRUE, details = TRUE),
          PAS = univ_quanti(pas, dig = 1, remove_miss = TRUE, details = TRUE))
     )
```

```
## $IMC
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0   24.5    3.1   15.4   22.3   24.6   26.4   32.5
##
## $PAS
##      N   mean     sd    min     Q1 median     Q3    max
##  300.0  137.1   16.8   92.0  125.0  138.0  149.0  177.0
```

Pour les variables qualitatives (dans leur format de facteurs `sexL` et `traitL`) :

```
with(df_1,
     list(SEX = cbind(N = table(sexL),
                      pct = round(prop.table(table(sexL)) * 100, digits = 1),
                      deparse.level = 2), # ajoute les noms de colonnes
          TRAITEMENT = cbind(N = table(traitL),
                             pct = round(prop.table(table(traitL)) * 100,
                                         digits = 1),
                             deparse.level = 2)))
```

```
## $SEX
##            N pct
## Féminin  153  51
## Masculin 147  49
##
## $TRAITEMENT
##                N  pct
## Placebo      120 40.0
## Traitement A  91 30.3
## Traitement B  89 29.7
```

4.4.2 Analyses bivariées

4.4.2.1 Comparer deux moyennes

Comparer la PAS en fonction du sexe :

```
### Tableau descriptif bi-varié
with(df_1,
     pas |>
       aggregate(by = list(sexL),
                 FUN = univ_quanti, # fonction à utiliser
                 dig = 1, remove_miss = TRUE, details = FALSE))
```

```
##    Group.1   x.N x.mean  x.sd
## 1  Féminin 153.0  132.4  16.8
## 2 Masculin 147.0  142.0  15.5
```

```
### Test de Student
with(df_1,
     t.test(pas ~ sexL))
```

```
##
##  Welch Two Sample t-test
##
## data:  pas by sexL
## t = -5.1382, df = 297.44, p-value = 5.029e-07
## alternative hypothesis: true difference in means between group Féminin and group Masculin is not equal
## 95 percent confidence interval:
##  -13.233087  -5.903634
## sample estimates:
##  mean in group Féminin mean in group Masculin
##               132.4248               141.9932
```

```
### On peut récupérer uniquement la p-value
# rappel : la p-value peut être récupéré avec l'opérateur dollar $
#          appliqué au résultat de la fonction t.test
with(with(df_1,
          t.test(pas ~ sexL)),
     p.value)
```

```
## [1] 5.028976e-07
```

```r
### Test de Levene pour vérifier l'égalité des variances
with(df_1,
     pas |> car::leveneTest(group = sexL))
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value Pr(>F)
## group   1  1.0178 0.3139
##       298
```

```r
### Test de Wilcoxon si les conditions d'application ne sont pas vérifiées
with(df_1,
     wilcox.test(pas ~ sexL))
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  pas by sexL
## W = 7609.5, p-value = 1.29e-06
## alternative hypothesis: true location shift is not equal to 0
```

4.4.2.2 Comparer 3 moyennes ou plus

Comparer la PAS en fonction du traitement :

```r
### Tableau descriptif bi-varié
with(df_1,
     pas |>
       aggregate(by = list(traitL),
                 FUN = univ_quanti, # fonction à utiliser
                 dig = 1, remove_miss = TRUE, details = FALSE))
```

```
##        Group.1   x.N x.mean  x.sd
## 1      Placebo 120.0  141.4  15.6
## 2 Traitement A  91.0  130.7  16.4
## 3 Traitement B  89.0  137.8  17.0
```

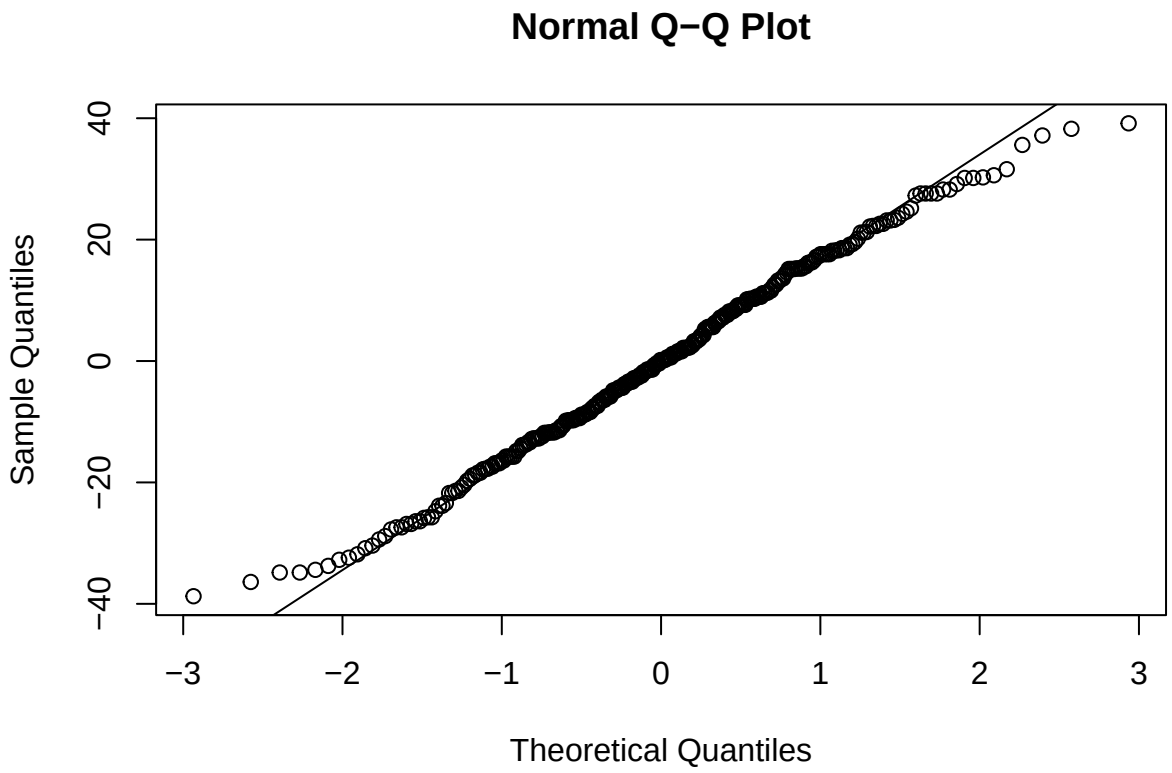
```r
### appliquer le test d'Anova au modèle linéaire pas ~ traitL
with(df_1,
     lm(pas ~ traitL) |>
       anova())
```

```
## Analysis of Variance Table
##
## Response: pas
##            Df Sum Sq Mean Sq F value    Pr(>F)
## traitL      2   5940 2970.18  11.215 2.016e-05 ***
## Residuals 297  78656  264.83
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```r
### Vérification des conditions d'applications
### 1) égalité des variances par le test de Levene
with(df_1,
     pas |> car::leveneTest(group = traitL))
```

```
## Levene's Test for Homogeneity of Variance (center = median)
##        Df F value Pr(>F)
## group   2  0.6669  0.514
##       297
```

```
### 2) normalité des résidus du modèle linéaire
res <- with(with(df_1, lm(pas ~ traitL)),
            residuals)
qqnorm(res)
qqline(res)
```




```
### Réaliser un test de Kruskal-Wallis si les conditions ne sont pas vérifiées
with(df_1,
     kruskal.test(pas ~ traitL))
```

```
##
##  Kruskal-Wallis rank sum test
##
## data:  pas by traitL
## Kruskal-Wallis chi-squared = 19.04, df = 2, p-value = 7.336e-05
```

4.4.2.3 Comparer des pourcentages

Description de la répartition du sexe en fonction du traitement, et application du test du Chi-2.

```
### Tableau descriptif bivarié
with(df_1,
     paste0(table(sexL, traitL),
            " (",
            round(prop.table(table(sexL, trait), margin = 2) * 100,
                  digits = 1),
            "%)") |>
       matrix(nrow = 2, ncol = 3, byrow = FALSE,
              dimnames = dimnames(table(sexL, traitL))))
```

```
##           traitL
## sexL       Placebo      Traitement A Traitement B
##   Féminin  "57 (47.5%)" "46 (50.5%)" "50 (56.2%)"
##   Masculin "63 (52.5%)" "45 (49.5%)" "39 (43.8%)"
```

```
### test du chi-2
with(df_1,
     table(sexL, traitL) |>
       chisq.test())
```

```
##
##  Pearson's Chi-squared test
##
## data:  table(sexL, traitL)
## X-squared = 1.5512, df = 2, p-value = 0.4604
```

```
# effectifs attendus :
with(with(df_1,
          table(sexL, traitL) |>
            chisq.test()),
     expected)
```

```
##           traitL
## sexL       Placebo Traitement A Traitement B
##   Féminin     61.2        46.41        45.39
##   Masculin    58.8        44.59        43.61
```

4.4.2.4 Corrélations

Estimation des corrélations de Pearson et de Spearman, avec test de leurs hypothèse nulles $\rho = 0$.

```
with(df_1,
     cor.test(imc, pas, method = "pearson"))
```

```
##
##  Pearson's product-moment correlation
##
## data:  imc and pas
## t = 3.8534, df = 298, p-value = 0.0001427
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.1072647 0.3231284
## sample estimates:
##       cor
## 0.2178593
```

```
with(df_1,
     cor.test(imc, pas, method = "spearman"))
```

```
## Warning in cor.test.default(imc, pas, method = "spearman"): Impossible de
## calculer la p-value exacte avec des ex-aequos
```

```
##
##  Spearman's rank correlation rho
##
## data:  imc and pas
## S = 3522826, p-value = 0.0001503
## alternative hypothesis: true rho is not equal to 0
## sample estimates:
##      rho
## 0.217141
```

4.4.3 Analyse multivariée

Estimation de la régression linéaire multiple de la PAS en fonction du traitement, ajusté sur le sexe et l'IMC :

```
with(df_1,
     lm(pas ~ traitL + sexL + imc) |>
       summary())
```

```
##
## Call:
## lm(formula = pas ~ traitL + sexL + imc)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -38.871 -10.083  -0.342  10.941  41.171
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)         109.5668     7.1403  15.345  < 2e-16 ***
## traitLTraitement A -10.9623     2.1246  -5.160 4.55e-07 ***
## traitLTraitement B  -2.8373     2.1374  -1.327 0.185388
## sexLMasculin         8.5634     1.7796   4.812 2.39e-06 ***
## imc                  1.1240     0.2904   3.870 0.000134 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.24 on 295 degrees of freedom
## Multiple R-squared:  0.1902, Adjusted R-squared:  0.1792
## F-statistic: 17.32 on 4 and 295 DF,  p-value: 8.911e-13
```